GENI Distributed Services

GDD-06-24

GENI: Global Environment for Network Innovations

November 16, 2006

Status: Draft (Version 0.7)

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 GENI Distributed Services Working Group.

Editors:

Thomas Anderson, *University of Washington*

Amin Vahdat, *University of California San Diego*

Contributing workgroup members:

David Andersen, *Carnegie-Mellon University*

Mic Bowman, *Intel*

Frans Kaashoek, *Massachusetts Institute of Technology*

Yoshi Kohno, *University of Washington*

Rick McGeer, *HP Labs*

Vivek Pai, *Princeton University*

Mike Reiter, *Carnegie-Mellon University*

Timothy Roscoe, *ETH*

Mark Segal, *Telcordia*

Ion Stoica, *University of California Berkeley*

We would like to thank David Oppenheimer, Larry Peterson, Jen Rexford, Steve Schwab and other members of the GENI working groups for helpful comments on earlier versions of this document.

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

Table of Contents

1 Introduction

GENI is an open, large-scale, realistic experimental facility that has the potential to revolutionize research in global communication networks. Its central goal is to *change the nature* of networked and distributed systems design: creating over time new paradigms that integrate rigorous theoretical understanding with compelling and thorough experimental validation.

The design of the facility is continually being refined to better accommodate research requirements and to leverage the best available technologies for its construction. Thus, the GENI design documents (including this one) should be seen as snapshots: a proposal for the design of the facility at a given point in time. Further changes and refinements are to be expected. We encourage the reader to make suggestions: the design described in this document is only preliminary and not final.

As context, we assume that the reader is familiar with the following documents:

- GENI Conceptual Design: Project Execution Plan [GDD-06-07]
- GENI Design Principles [GDD-06-08]
- Overview of the GENI Architecture [GDD-06-11]

All of these documents are available on the GENI project web site: www.geni.net

To develop the project plans from a Conceptual Design to a Preliminary Design, the GENI planning group has formed six working groups, with roles described below:

- Research – refine the rationale for the facility and the requirements the facility must meet to support its intended use
- Backbone – refine the design of the wireline components of the facility, including backbone network technologies and programmable routers
- Wireless – refine the design of the wireless and sensor network components of the facility
- Facility architecture – refine the overall architecture of the facility including the interface between wired and wireless components and higher level distributed services software. This group is also responsible for ensuring that the facility as a whole meets the requirements defined by the research working group.
- Distributed services – refine the design of a set of distributed services to achieve the high level goals of the facility. This group is also responsible for specifying the configuration of the flexible edge cluster.
- Education and outreach – develop a plan for how the facility can be used for education and broader community participation.

The goal of this document is to describe the requirements and design for the distributed services part of GENI; this document is intended to complement the design documents being produced by the other five working groups.

One way to understand the boundary between the distributed services and other groups is that the backbone, wireless, and facility architecture groups are primarily chartered with designing GENI so that it can support a wide spectrum of cutting-edge network and distributed systems experiments; by contrast, the goal of the distributed services group is to not to make the facility more powerful, but rather to make the facility more *usable*.

We identify six separate user communities:

- Researchers. This is the principal user community for GENI, and our goal is to remove as many practical barriers as possible to this community making full use of GENI. A typical network or distributed systems research project is conducted by a single principal investigator along with a single student. For GENI to be practical for these users, the overhead of understanding how to map their intended experiment onto GENI must be within reach – ideally, requiring at most a few hour tutorial to get started. This means we need to provide a rich set of tools for configuring, monitoring, and debugging experiments, a rich set of common utilities to be used by experimenters, and predictable and repeatable behavior for experiments running on the system. At the same time, GENI will also need to provide access to the full set of capabilities of the system for "power users".

- Operations staff. Any facility of this size will require full-time staff to manage the system and keep it running. NSF rules imply that the operational cost of the facility will be borne by NSF's research budget, and thus we need cost-effective tools to reduce the administrative cost of managing the facility: maintaining security of the facility, keeping software up to date, upgrading hardware, adding new users and managing their privileges, etc.

- Resource owners. Resource owners include NSF and any other organization that might contribute resources, such as the European Union or even individual universities hosting parts of GENI. The principal goal of NSF in building GENI is to foster revolutionary research in network and distributed systems design. To this end, NSF (or its community proxy) will need policy knobs that allow for explicit decisions about relative research priorities when there is resource contention between experiments. Similarly, the international visibility of the facility requires care in ensuring its secure operation. Note that NSF is unlikely to be the only resource owner. If GENI is to be successful over the long term, we must be able to accommodate new hardware resources being contributed by the community. For example, as architectures and services running on GENI become more successful, their resource demands are likely to outstrip whatever NSF can afford; we should design to gracefully handle this success scenario. This means we need knobs for a broad set of resource owners to control how their resources are being used in the GENI context, if GENI is to be sustainable over the long term.

- System developers. Some of the software services described in this document are at least partly intended for internal use, to make it simpler and less expensive to develop the facility itself. But as with hardware, software being constructed for GENI is only the beginning of the story. Each hardware and software component in GENI, if constructed as a commercial product, would likely outstrip the entire GENI budget by itself. Thus GENI can only fund the beginnings of a workable system. If GENI is to be successful, we also need to create an ecosystem where system developers – including researchers, but also potentially including the commercial software development community -- will be able to contribute software to further enhance the GENI platform.

- End users. A key motivation for GENI is that network and distributed systems research ideas need real-life validation by real users. These users can be the research community "eating its own dog food", but there is a larger community of non-expert users eager for a more secure, robust and functionally advanced Internet. For outreach and to improve the quality of the research done on the facility, we want to make it easy for normal users to access and benefit from the advanced technologies being tested and demonstrated on the facility, for example, directly from their desktop.

- External parties. As a platform for experimental research that is designed to interoperate with today's Internet, experiments are likely from time to time to impact the external world, potentially causing either perceived or real harm to third parties. Operational procedures, along with support software, will need to be in place to minimize the likelihood of negative impacts, while also providing the tools to quickly diagnose and deal with complaints. A final set of users is worth keeping in mind: malicious attackers. Due to its prominence, GENI may from time to time come under determined attack, either to subvert GENI resources to some other end, or simply to deny access to GENI resources to researchers or the general public. The facility must be designed to prevent such misuse, while still providing a system that is as easy to use as possible for the other user communities described above.

Collectively these users add a set of practical design goals beyond what is described in "GENI Design Principles" [GDD-06-07]. Specifically the distributed services working group is concerned with:

- Security and isolation. It is essential that the facility be secure from malicious attack, and that experimenters be able to use the facility without undue concern for what other experiments are simultaneously running on the facility.

- Operational cost and manageability. It is essential that the facility be manageable and operable once fully constructed.

- Usability and flexibility. It is essential that the facility be usable by both researchers and end users, and that the facility avoid unnecessary limits on the types of research it supports.

- Experiment development cost. It is essential that the cost of developing a new experiment to run on the facility be within the typical budget of a network and distributed systems researcher.

- Construction cost and development schedule. It is essential that it be possible to construct the facility for the allocated budget. This implies, for example, using off the shelf software and hardware components where possible, and only undertaking to build GENI-specific engineering efforts where there is significant leverage.
- The first of these concerns, facility security, is treated in depth in a companion document, “GENI Facility Security” [GDD-06-23]. The rest of these concerns are the focus of the document you are now reading.
- Finally we should note that the GENI design is a work-in-progress, and in fact its design is likely to keep evolving even after construction begins and the system begins to be used. This has two consequences for our design plans. First, it is our intent to “build-use-iterate”, Second,

2 Document Structure

To address these varying sets of needs, we have split our discussion into several sections. In Section 3, we describe the edge device hardware and software: what computational and storage resources will be available at each location for running experiments? In Section 4, we discuss storage, an essential element of many proposed experiments on the facility and also a key element in other parts of the distributed services design. In Section 5, we discuss resource allocation – how do we arbitrate among the competing demands of researchers for scarce GENI resources? In Section 6, we consider experiment support – how do we make it easier for novice users to launch experiments? In Section 7, we focus on operations support, and in Section 8, we discuss the support for legacy applications and users needed to drive real use of experimental GENI services.

It is our goal to lay out a complete design for the distributed services part of the facility. However, to paraphrase a military maxim, no system design survives initial contact with the user. Experience with real use is essential to guiding the design of any real system. We expect the requirements described in this document to be continually refined up until and indeed past when construction begins. Our strategy is to “use while we build” – to design a system whose initial implementation (albeit with limited functionality) can be quickly deployed to guide the relative priority of additional capabilities and features based on user feedback. GENI in this respect is no different from any other large software development project. Similarly, it is likely that the policy decisions made about GENI’s use are also likely to evolve over time and with use. Thus a goal of our design is to decouple, as much as practical, the system’s architecture and artifacts from specific policy decisions, to allow policy to evolve separately from the architecture of the facility.

Please note that this particular document is a very early draft of the preliminary design; the draft is incomplete, mixes concerns that should be separated, contains numerous open issues, and completely ignores many essential topics. Further, the text is uneven: different sections of the design have progressed to different degrees of completeness: some have only high level requirements, some have requirements and an initial design, and so forth. Much more detail is needed in virtually every area of the document. We expect to address the limitations of this draft over the next few months and years, but we reiterate that we welcome any feedback the reader is willing to offer.

Eventually, each section of this document should address the following questions:

- Motivation. What is the problem being addressed and why is a solution essential to GENI's success? Who are the users being helped, what do they care about, what are their requirements?

- Approach. What is the technical design proposed to address the problem and why is the technical solution the most appropriate choice. What are the positives and drawbacks to the approach? What is the related work and what are the lessons to be learned from it? What are the open issues? prior work;; open issues

- Usage scenario. How would the approach work for a typical user?

- List of work items. What are the specific pieces of software or hardware that will be purchased, modified, or constructed? What existing artifacts can be leveraged by GENI in whole or in part?

For each work item to be constructed, an additional list of questions must be addressed. Following the X.900 standard:

- The enterprise viewpoint: the function of the work item, but not how it functions. What the system is for, and what it is trying to achieve.

- The information viewpoint: the conceptual ontology of the information used by the system.

- The computational viewpoint: software objects and interfaces between them.

- The engineering viewpoint: issues of performance, reliability, availability, security, and distribution.

- The technology viewpoint: Technologies used to construct the system. What is the sequence of milestones or functionality to be rolled out over time; what are the dependencies -- specific pieces of software or hardware that will need to be constructed before work can be completed? What is the schedule and budget, and what are the most important risk factors?

3 Flexible Edge Cluster

Clusters of commodity PCs will be the workhorse nodes in GENI. They provide the computational resources needed to build wide-area services and applications, and even in situations where special-purpose hardware is more appropriate, general-purpose processors will allow researchers to work on the functionality of new architectures while these new technologies are developed and hardened. To be specific, our plan is that PC clusters will be distributed to 200 edge sites at major universities around the country, where they will host overlay services and serve as “ingress routers” for new network architectures. We will vary the size and unique capabilities of these clusters as user demand dictates, for example, by adding large storage capacity to a subset of the edge sites. Many interesting future applications are likely to involve large volumes of distributed storage.

One question we need to address is how much GENI hardware do we need at each site? One answer to this question is that no amount of hardware will be sufficient! Virtually everyone in the US will be near some GENI edge site, and so if GENI is successful in fostering novel network architectures or distributed services that become widely used, demand for those services will likely outstrip any hardware we might provision in advance. This is the classic "success disaster" scenario. Thus, we advocate provisioning modest amounts of hardware to handle foreseeable demand, but also making it easy for universities and other interested parties to add resources into GENI. We envision deriving the specific GENI hardware requirements from the list of candidate experiments being collected by the Research Working Group. In the meantime, as a back of the envelope calculation, a typical PlanetLab site has two PC's, but these have nearly full utilization of CPU, memory and disk storage. Many researchers find PlanetLab too busy to be of practical use for running their experiments. Since GENI is intended to support a much broader research community than PlanetLab does today, we advocate substantially increasing the amount of resources per site. Specifically, we propose placing a cluster of approximately 10-20 servers at each site, each server with a modern multicore processor, 8 GB of RAM, 1 TB of disk, and gigabit Ethernet. A high speed switch would connect the servers together and from there through an edge router to the rest of GENI. Some advanced experimental distributed services will need more computation, memory, or disk storage than a single site might provide; for these, we envision putting much larger clusters at a smaller number of sites, e.g., 20 sites spread around the country would have 200 server clusters.

We assume each edge cluster PC will be configured with trusted computing (TC) hardware likely to be standard in the next few years. This hardware will allow GENI to remotely ensure that each node is running a known version of a specific operating system, and that the machine can be rebooted to a known state (or in the case of a newly discovered vulnerability, to a new version of the operating system) on demand. PlanetLab uses a version of this approach, by initially booting the system off a CD-ROM containing a private key specific to the node, the public key of GENI central, and a boot loader that securely fetches from a central repository the most up to date version of the operating system for that PC. Native TC hardware support adds one additional layer of security, in that unlike a CD-ROM, the TC hardware cannot be corrupted or changed without extensive, obvious, and technically challenging physical tampering with the machine. Note that we can fall back to using a CD-ROM or dongle if the planned transition to TC hardware does not happen in time.

Using a secure boot loader, each PC in the cluster runs an operating system that supports one or more virtual machines, each of which is bound to some subset of the PC's memory, disk, CPU, and network capacity. As such, the PC is like any other element in GENI, and must follow the interface specification described in the "Overview of the GENI Architecture" document. Briefly, each experiment that wishes to run on such a node is allocated a virtual machine on one of the PCs. A virtual machine is programmable in the most elemental sense, in that a researcher could create and run one or more computer programs in each virtual machine in its slice. Researchers are provided maximal flexibility in that they can write their own software from scratch inside the virtual machine, using conventional programming languages. At the same time, the software implementing the virtual machine provides a layer of isolation between experiments and a layer of protection against misbehaving experiments. For example, the virtual machine could prevent the experiment from sending spoofed packets into the Internet, unless explicit permission to do so was granted by GENI administrators for some well-justified purpose. A more complete discussion of GENI security issues is covered in a separate document [GDD-06-23], and we defer to Section 6 a discussion of the support we propose to provide experimenters, in terms of automating some of the mechanics of developing and deploying candidate architectures and distributed services on GENI. We also envision that researchers and others would contribute their own software modules to GENI, to provide services that others can use to build their prototypes.

There are several candidate operating systems we could run on the PCs in the edge cluster. One possibility is to use a minimal virtual machine such as Xen or VMware. These systems provide maximal flexibility in that they provide the abstraction of the raw hardware to each experiment. They also provide strong isolation between experiments, as dedicated hardware resources can be assigned to each experiment. A drawback is that these systems come with a very limited programming environment (essentially, experimenters would be given little more than virtualized bare hardware). Since many experimenters do not need low level hardware access on the edge PC's, and because providing such low level access is relatively expensive (because memory and CPU cannot be shared across experiments), we envision also providing experimenters the option of running in a virtual machine on top of a normal operating system, such as Linux vservers. Linux vservers is a modified version of Linux that provides each experimenter the illusion of a virtual dedicated Linux PC with root permission. This way, the experimenter can use all of the facilities of Linux, yet still have complete control over the virtual machine as if they were the only user of the machine. (Of course, under the covers, the physical PC is being shared between multiple experiments and code inside the Linux vserver implementation prevents experiments from exceeding their privileges, e.g., by preventing the virtual Linux from sending spoofed packets.) Virtualizing above the operating system is a more efficient option since the software implementing the Linux vserver environment (a variant of Linux itself) can be shared across experiments. Linux vservers have the corresponding downside that they have relatively weak isolation between experiments; although some research operating systems have been built that provide strong isolation between processes running on the system, almost all widely used operating systems lack strong isolation. The result is that an experiment can be negatively impacted by the behavior of other experiments, e.g., through memory thrashing, file descriptor exhaustion, disk sharing, etc.

We do not believe it necessary or advisable to pick a particular operating system for the GENI edge cluster at this date. However, we do believe that virtualization at both the minimal virtual machine layer (as in Xen) and at the operating system layer (as in Linux vservers) is a practical requirement. Experimenters needing tight control over their resources would choose to operate closer to the bare hardware, while others tolerating a best effort approach to resource isolation could benefit from the richer set of services provided by a full operating system. In either case, since no existing operating system implements the full set of functionality called for by the GMC specification, we will need to make some modifications to whatever operating system or systems we chose. Because developing or maintaining a full operating system could consume the entire GENI budget, to little practical benefit given the enormous private industrial investment in operating system technology, we envision keeping these modifications minimal and easily folded back into the original operating system code base, so that we can track external improvements in the underlying operating system code.

As described in the “GENI Component Reference Design” [GDD-06-13], to support the abstraction of a virtual machine, the operating system running on the underlying computer must (1) allocate and schedule resources (e.g., CPU, bandwidth, memory, and storage) so that the experiment’s requirements are met and the runtime behavior of one slice does not adversely affect the performance of another running on the same node, (2) partition or contextualize the available name spaces (e.g., network addresses and file names) to prevent a slice from interfering with another, or gaining access to information in another slice, (3) provide a stable programming base that cannot be manipulated by code running in one slice in a way that adversely affects another slice, and (4) enforce (and log for audit) access control decisions with respect to the specific permissions granted to a slice, such as the ability to send packets to the Internet vs. only to other GENI nodes.

Virtual machines are now a mature technology and they have proven effective in supporting experimentation with distributed services in PlanetLab [BAV04, BAR03], making them a natural, low-risk starting point for building nodes for GENI. Today, packet forwarding within a virtual machine can forward data packets at nearly 1 gigabit/second. As GENI evolves, we can move key functionality into the operating system for faster throughput and provide an increasingly fine granularity of sharing of the system resources by exploiting hardware support for virtualization [IVT].

One issue raised by the use of virtual machines is addressability. To the outside world, and specifically the rest of the Internet, the edge cluster PC appears to be a normal PC capable of sending and receiving packets in the normal way. However, experiments running on a given node need to share this external view – two web servers, each running on their own virtual machine on the same physical hardware, cannot both receive all packets arriving on port 80. We discuss this issue in more depth in Section 8, as one solution is to provide a more dynamic name system to allow more flexible binding visible to the outside world. Here we note that a pragmatic solution is simply to allocate an ample number of IP addresses to each cluster, e.g., a \24 (256 addresses) for a 10-20 node cluster. Each PC can then have multiple addresses, enabling it to support a limited amount of unambiguous multiplexing on each port. The converse issue is that many types of fault tolerant network and distributed systems experiments need to have facilities for a single address to be mapped to a set of physical hardware, so that a failed node can be transparently hidden from view. For transparent failover within a cluster, this requires the ability for virtual machines to manipulate the physical ARP tables in the cluster switch (if a controlled, secure fashion); for transparent failover across clusters, this requires BGP connectivity. Both issues are discussed in more depth in Section 8.

A final issue concerns cluster-wide operating system services. Certain experiments may need coordinated behavior across multiple machines in a given cluster and/or to bind themselves to a particular node within a cluster. To accommodate these requirements, we believe the facility architecture specification – the primitives supported by every component in GENI -- needs to be rich enough to be able to capture these constraints. We defer a discussion of all other cluster-wide services, such as cluster-wide reliable storage service or cluster-wide resource allocation, to later sections in this document.

4 Storage

GENI requires several forms of storage to facilitate both management and experimentation. Experimenters will need local storage on the nodes they use (e.g., to store results, logs, or the data their experiments operate on). Experimenters will also benefit from convenient remote access to those storage resources, to debug their experiments or examine results. Researchers will need higher performance storage resources within a cluster of GENI machines for the data that their experiment operates on (e.g., cached web pages, network measurement data, and so on). Finally, the management of GENI will require significant quantities of reliable storage for audit trails, resource use accounting, and tracking GENI resources such as equipment and network connectivity. Section 5.1 explores the goals and storage requirements in more detail, and the subsequent sections examine the resources and technical developments needed to meet those goals. As we will see, the storage services are dependent on the security and resource-allocation designs.

4.1 Storage goals and requirements

This section lists the requirements for storage services on GENI. We expect GENI to provide a large amount of data storage in the form of many disks. These disks maybe attached to end nodes or directly accessible over a local-area network (i.e., network-attached storage devices). We call nodes that have storage attached to them storage nodes, and we expect them to be part of the edge clusters described in Section 2.

The storage nodes will have different users: 1) the GENI operational staff who need full access to storage on each storage node and the ability to manage remote storage nodes easily; 2) distributed system researchers who may produce new storage services; and 3) researchers who produce new GENI experiments that may involve storage (e.g., to log experimental data). To support these different categories of users we envision a number of different storage services with different interfaces.

Goal #1: Provide sufficient storage to support the users and managers of GENI.

We envision that many applications and experiments that run on top of GENI will require significant quantities of storage.

- An important goal of the resources provided by GENI is to be able to perform research that is valid in the context of both today’s and tomorrow’s industry. With corporations such as Google, Akamai, and Amazon.com having tens to hundreds of thousands of nodes, GENI experimenters must have access to sufficient resources to provide convincing evidence that their solutions will scale to the sizes that industry demands and will demand in the future.
- To estimate the storage requirements for GENI, we extrapolate from both current industry capabilities and experiments running on current testbeds such as PlanetLab. Examples of those experiments include:
- MIT’s OverCite project uses roughly 40 GB per node, a requirement that makes it very difficult to run on PlanetLab.

- CoDeen uses 3Gbyte per node.
- Coral uses 4 Gbyte per node, but would benefit from 10 Gbyte per node.

Akamai Inc. claims to use more than 20,000 servers for content distribution and Amazon's Inc. EC2 service allows each user 20 instances of a distributed application, with about 160Gbyte disk space for each instance.

Based upon these numbers, we expect that GENI should have at least 10,000 servers, with an aggregate storage capacity of 20 petabytes using 2006 disk technology; we expect these numbers to increase commensurate with improvements in disk technology as disk space is added at later stages of GENI development, and as the demands of experiments increase.

Goal #2: Provide access to the storage resources.

Experimenters and GENI administrators must be able to quickly and easily access the storage resources provided in the infrastructure. This includes: (1) direct access by experimenters running code on an individual machine; (2) access through databases for more convenient access to structured data storage; (3) convenient remote access for experiment and system management.

On a particular node, researchers expect a convenient, file system-based interface to read and write the data involved with their application. Past this, however, many experiments will benefit from a more structured, database-like access mechanism to, for example, store and query data from experiments. Because of the utility of this functionality to the management of the GENI nodes themselves, we believe that the basic node interface should include a local database-like access mechanism.

Both experimenters and the GENI managers need to remotely access data stored on GENI nodes. Experimenters must distribute code, gather analysis results, and may need to interactively access data while developing and debugging systems. GENI administrators need remote access to analyze problems, failures, and monitor the performance of the system. To support these needs, GENI should provide a convenient mechanism for remote data access. This access has two aspects: First, a remote file system-like access mechanism that provides easy, human-friendly remote access, perhaps at the cost of some performance. Second, a high-performance data interface that can be a building block for the development of other services; we discuss this need further in the next goal.

Goal #3: Provide high performance storage for experiment support.

Services that run on top of GENI may need to rapidly read and write large volumes of data to storage. At a minimum, GENI must provide access to high performance storage on individual nodes, and make it possible for services to services to leverage the parallelism available from having many machines with many disks in a single location. It is, however, equally important to ensure that the services provided by GENI are feasible to implement in a reasonable amount of time, and do not excessively specialize to a particular application's need. Therefore, we believe that GENI should meet three requirements to satisfy the needs of a diverse array of services:

First, GENI should allow services to rapidly and efficiently access storage resources on a node both when they are running locally on that node and when they are remote. This access could take the form of file system access for local processes and as a simple block-based mechanism for remote access. We explicitly note that to meet this requirement, GENI need not provide a high-performance wide-area file system.

Second, for the convenience of service developers, GENI should provide a high performance clustered file system for use within a single, geographically co-located collection of nodes. Local clustered file systems are well understood within the high performance computing community, and can be reasonably provided within GENI without incurring undue risk of failure. Many services (e.g., caches such as CoDeeN and Coral) will benefit from having a more reliable, high performance clustered file system without having to re-implement this functionality.

Finally, to support the development of alternate clustered or wide-area file systems or storage systems, GENI should make it easy for services to access the parallelism available on the machines in its clusters. This support implies two capabilities: First, the ability for services to determine which resources can operate in parallel (e.g., to differentiate different physical disks from virtual disks exported by the same hardware), and second, the ability to access these different devices quickly and in parallel.

- Goal #4: Balance cost, performance, and the expectation of durability.

One primary threat to storage is that disks may fail unexpectedly and lose their content. Providing durable storage under this threat is challenging and it is helpful to make a clear distinction between hard-state and soft-state data. Hard-state data must be durable, perhaps through replication and archiving. Soft-state data can be reclaimed at anytime, perhaps, if possible, after warning the owner of the data. We envision that GENI should enable hard-state and soft-state services, but GENI itself should provide at a low level only “best-effort” durability.

Best-effort storage is defined by three characteristics: First, GENI will allocate long-term storage to projects when necessary, and not reclaim the storage until that time has expired. Second, the GENI management must create and follow guidelines about the frequency of intentional storage erasures (e.g., due to upgrades or routine maintenance). Finally, services must use replication or other strategies to ensure that they can survive accidents or erasures within the GENI parameters.

Hard-state services are more challenging to develop and maintain than soft-state services. We don’t assume that the GENI operational staff will be directly responsible for archiving and replicating hard-state data. Instead, we assume that the GENI operational staff allocates disk storage to a service for a long period time with the promise not to reclaim that storage for that period. The service may then use the assigned storage to provide durable hard-state data. (If such a service becomes popular, the operational staff may choose to take on the responsibility of running the service.)

- This is a combination of management promises (not wipe all machines at the same time, etc.) along with constraints about the actual hardware used in GENI (should meet certain levels of reliability) and making clear the "contract" between researchers and the GENI storage services about how researchers should protect their data from accidental erasure.

- GENI experimenters are responsible for performing whatever duplication of their data is necessary to provide the level of durability they require. We envision, however, that services will arise that provide this duplication on their behalf. The GENI architecture should be conducive to the creation of such services.
- Goal #5: Allow experimenters to build new storage services through extensibility
- A primary purpose of GENI will be supporting experiments in distributed computing. These experiments will test novel distributed applications that aggregate the storage at many GENI sites. As demonstrated with the experience of PlanetLab, some of these experimental applications will result in new services that will be used by the GENI community. To facilitate such new services, it is important that the storages services can be easily extended. For example, it should be possible for a researcher to build a novel file system by extending the low-level node interface and still achieving good performance. Another researcher may want to extended storage services at a higher-level, for example, by extending the cluster file system.
- A modular storage design also reduces the pressure on the GENI operational staff in several ways. The operational staff must provide basic storage services but doesn't have to build all services; the staff can rely on the community to help out. The operational stuff doesn't have to run all services but can rely on the experimenters to run their services---they will have a vested interested in doing so to be able to collect interesting experimental data with real users. Finally, the operational staff can evaluate multiple storage services and select the best one for the GENI community, and grow the services in an incremental fashion.
- Goal #6: Strong user authentications and access control
- Since storage is durable and may contain valuable information, it is important that the storage services provide strong security. When a user makes a request for storage, the storage allocator should be able to authenticate so that services can make a decision if the user should be granted the request or not, and can audit the use of the storage over time. Storage services will storage important data; the owners of the data must be able to control which users can access it. To support such security policies, the storage services need an infrastructure for user authentication.
- The rest of the section describes a storage design that can facilitate these goals.

4.2 Per-node storage services

On each storage node, GENI should provide standard storage interfaces, which include a file system interface, a raw disk interface, and a database. The file system interface is primarily intended to manage local storage and organize it with existing tools. The raw disk interface allows system and application developers to implement new storage services, without having to pay the overhead of a file system.

We expect that the file system and raw disk interface will be used to build all other storage services. This implies that the file and disk interface should provide good performance. That is, the performance of the other storage services should be bottlenecked by disk and network transfer rates, not by the implementation of the file and disk interface, and its interaction with the operating system. In particular, this requirement calls for good support to overlap I/O and computation, and careful scheduling of the storage services with the programs that use them.

We expect that each storage node runs two other services by default, because we expect that these two will be used frequently. The first one is a standard SQL database. Many services and applications are likely to need a database that can lookup items by key; the database's performance will be crucial to these applications. For example, the GENI monitoring services need an SQL database, and GENI might as well make the database used for monitoring available to other services too, instead of forcing users to tune such the database, which can be complicated. Given that databases today provide a notion of user, it should be reasonable simple to incorporate the database into the GENI authentication and authorization design.

The second service is one that allows system developers to isolate applications and provide new storage services by intercepting file system calls. This can be done in various ways, but two candidates ones are through a file system library and a "loop-back" file system. Using this service, different kinds of distributed services can be added without having to change the software on the individual storage nodes in fundamental ways.

4.3 Distributed data services

With the above storage node services, we expect that system and application developers can construct many distributed storage services. Many users, however, will have a need for a common set of services. These include the following:

- A wide-area file system. This service is mostly intended for system administrators so that they can manage remote nodes easily. A distributed file system extends the familiar local file system interfaces across all storage nodes and allows all storage to be accessed through global pathnames. Possible candidates include AFS, NSFv4, and SFS.
- A high-performance wide-area file system that is suitable for reading and writing files by wide-area applications that require high performance, but perhaps can compromise on other file system functions. An example of such a service is a rudimentary read/write segment-store service. This service allows a program running on any storage node in GENI to create and access big extents on any other storage node. The extents are named by unique identifiers. This segment store gives system developer and researchers control of where data will be stored, allows remote access, and is a building block for many distributed storage services and applications. The Google file system library may be a good starting point for the design of this service, but more ambitious designs that provide a higher-degree of durability and transparency need to be explored.
- Write-once or read-only storage services. These services allow efficient access to data that is read-only or write-once (e.g., a soft-state CDN and a hard-state DHT). Because

these services are targeted to specific usage patterns, these services can provide better performance than a distributed file system. These services allow programs and users to have access to large data efficiently through caching, proximity-aware retrieval, and parallel fetch.

- Data push services. These services push data from one storage node to many nodes. Push services should allow users to distribute programs efficiently, to run a new experiment, restore data on remote storage nodes, etc. Possible designs include CoDeploy, BitTorrent, Shark, and dot.
- Logging services. The storage service should provide good support for event logs. Distributed applications should be able to log events, collect them, analyze them, and roll-over logs easily and at high performance. The storage service should log events about itself. Events include storage went off-line, disk was lost permanently, etc

4.4 Securing and managing storage

Many applications will use the storage services and it will store valuable data and be one of the primary resources of GENI. The amount of storage in GENI will be large and cannot be easily reshuffled from one to another. It is therefore important that the storage services provide good support for securing and managing storage for system administrators, developers, and users.

The storage service should provide some level of security. At a minimum, it should protect against other user's mistakes (e.g., some user typing "rm -rf *" by accident and over use (e.g., through quota). The security support will leverage the distributed authorization and authentication services discussed in the previous section. Protection against over use is part of a larger plan for resources management, which is discussed in another section.

The requirements and the design of this aspect of the storage services are dependent on the GENI security and resource allocation plans. Storage services should be a primary test case for these plans. As these plans develop, the design for securing and managing storages will be made more specific.

4.5 Construction Sequence

- Develop segment store on available compute nodes Depends on VMM fast raw disk partitions.
- Clustered filesystem: immediately after segment store is working.
- Management" wide-area filesystem: concurrent w/segment store. Requires loopback filesystem interface.
- Global read-only/mostly filesystem. Local VMs/vservers access this via the loopback interface. Could be written atop the local filesystem (concurrent w/segment store). Alternatively, could be written to use the segment store or the clustered filesystem. If using clustered filesystem, then develop local filesystem version first.
- Global high performance read/write filesystem:

5 Resource Allocation

Resource allocation is key to any shared computing and communication infrastructure. In particular, GENI will be shared by thousands of different organizations and tens of thousands of simultaneous users. While there will be significant resources available system-wide, GENI will undergo a ``tragedy of the commons"---where the system loses value for all participants---if all users are allowed to greedily consume as much resources as possible. Such scenarios take place periodically on PlanetLab but will be unacceptable for some GENI users, for example those wishing to continuously run a Virtual ISP in a Slice. First, such services will typically have more demanding resource requirements than a typical PlanetLab experiment. Second, such services may need a baseline level of guaranteed resources to accurately model real-word deployments. For example, ISP's can usually assume that any resource scarcity is caused by their own behavior rather than the behavior of those sharing the physical underlying resources. At the same time, some services may wish the validate their ability to run under with only best effort guarantees. Further, employing best effort resource scheduling usually results in higher levels of resource utilization. Hence, GENI resource allocation should support both guaranteed and best-effort resource allocation models.

Existing shared testbeds either perform strict space sharing, where individual jobs control a subset of nodes through completion as in Emulab or Deter, or best effort scheduling, where individual jobs receive resources inversely proportional to the number of currently competing jobs as in PlanetLab. However, to achieve both high system utilization and to allow at least some applications to run with dedicated resources, we propose that some subset of global GENI resources will be subject to admission control at a fine granularity, i.e., dedicated access to portions of multiple machine's CPU, memory, network, and storage resources. The global GENI resource allocation mechanism will interface with the per-node virtualization layer to effect appropriate resource isolation. By default, most applications will run in "best-effort" mode (perhaps with some priority relative to other applications) to ensure a high-level of resource utilization and to reduce the barrier to entry for running jobs on GENI. The amount of resources that can be reserved by an application, the relative priority of best-effort applications, and the proportion of global resources subject to admission control are all matters of policy.

An important task of the resource allocation mechanism is to encourage efficient resource usage, e.g., to ensure that users only acquire as much resources as they truly need, and for only the period of time that they really need them. This second point is particularly important. The duration of a particular resource binding should be coarse-grained to reduce the overhead associated with performing resource allocation (i.e., we wish to avoid requiring an application to renew its resource privileges every few seconds). However, we believe that resource re-negotiation must take place more frequently for applications as they are dedicated increasing amounts of resources. For instance, an application with dedicated use of a substantial fraction of global GENI resources may need to renew its resource privileges every few minutes (to allow the system to be agile to changing resource demands). A best effort application running with low priority may only need to re-negotiate on the granularity of days or even weeks. In general, GENI's resource allocation mechanism should be general to support a variety of policies for the duration of resource bindings, but guaranteed resource allocations must always be time-limited.

The resources allocated to individual services will often be for shorter durations than the lifetime of many GENI services. Thus, many GENI services should be prepared for dynamically changing levels of available resources independent of whether they are running with resource guarantees or in best effort mode. To allow applications to cope with a constantly changing resource landscape, the resource allocation mechanism should support callbacks to the service (or designated agents of the service), similar to the techniques employed by Exokernel or Scheduler Activations.

5.1 Goals

Given the centrality of resource allocation to GENI, below we describe some of the high-level requirements of any such system.

- *Heterogeneous user base.* Applications will range from network experiments (measuring some aspect of GENI or the Internet at large) to long running services (for instance, content distribution services such as CoDeeN or Coral) used by a large number of external users to fundamental research into network architecture, for instance, a virtual ISP transiting data transparently to and from destinations unaffiliated with GENI. In all of these cases, we require some mechanism to divide global system resources---CPU, network bandwidth, memory, disk storage---among participants. Resource allocation requirements include: interactive access to CPU and network resources to support network measurement, some baseline guarantee of (relatively large) resource availability over coarse time scales to support long-running services, and dedicated access to resources to support virtual ISPs. While there has been research in supporting all of these resource allocation models, there is no system that supports all such uses in one framework. For example, any such system would require advancement in both local and global resource scheduling mechanisms.

- *Resource allocation policy*. Even in the presence of appropriate resource allocation mechanisms, a primary question remains: the appropriate policy for making specific resource guarantees to individual users and applications. In a system with *n* users, should GENI allocate *1/n* of the resources to all participants? This might encourage certain users to consume resources even when they are not deriving significant benefit from those resources. Should there then be appropriate incentives for being a good citizen, e.g., for backing off resource consumption during times of global resource constraint? What are the appropriate incentive models? For example, in exchange for lower overall rates, certain companies agree to reduce power consumption during times of peak demand. Finally, should there be some central authority that decides on baseline resource allocation levels? For instance, a popular virtual ISP that is supporting basic research in next-generation Internet architecture perhaps should be allocated more than *1/n* of global resources. There are interesting questions in who sets such policy and the process for making these decisions.

- *Resource isolation*. Given some policy for allocating resources among competing principals, we require techniques to ensure isolation among competing tasks. For instance, if a virtual ISP is promised some level of bandwidth and computation at a set of nodes spread across GENI, then those resources must be available independent of what other users are doing. One natural model for providing such isolation is through virtual machines. Applications can then configure individual virtual machines appropriately while a virtual machine monitor running on each host enforces resource allocation and isolation on a per-node basis. One interesting question is whether resource allocation guarantees should be made globally or on a per-node basis. For example, should a user be promised 30% on each of 100 machines spread across GENI or the equivalent of 30 machines worth of resources GENI-wide (to be sub-divided dynamically).

- *Security and authentication.* Security is a primary requirement for any resource allocation infrastructure. In particular, users must have a secure mechanism for authenticating themselves to the system and to receive the resources that they are entitled to. Users should not be able to consume more than their allocated share of resources and, just as importantly, should not be able to deny other users from consuming their allocated resource share. For generality, the system must support secure capabilities, the ability to transfer to a second party resource privileges originally assigned to some first party. Such capabilities should be self-describing, unforgeable, traceable, and irrefutable. These capabilities can form the basis of powerful resource peering mechanisms and policies to allow decentralized access to global system resources.

- *Separation of policy and mechanism.* We aim to provide the mechanisms required to implement a wide range of resource allocation policies, many of which we cannot anticipate. For example, while we dictate that users will be granted best-effort or guaranteed-share access to resources, we do not want to constrain policies about which users are granted access to which resources, the relative priorities of user jobs, how users decide how much resources to request and use, etc. On the other hand, we must define basic mandatory classes of resource principals and types of capabilities (access rights) that will ensure that regardless of resource allocation policy, users will not be able to exceed their authorized resource usage. These mandatory pieces of the resource allocation infrastructure will also allow different resource allocation policies to peacefully coexist on top of the same underlying resource allocation mechanisms.

- *Distributed architecture and local autonomy*. GENI's resource allocation architecture should be logically decentralized to support both federation and local site autonomy. However, initial implementations of the architecture may be logically centralized for expediency. As part of the requirement for providing sites with proper incentives to consider global system conditions when making local decisions, sites contributing resources to GENI must in the end maintain autonomy over their own resources. Even if GENI is initially controlled centrally, it must eventually support federation with other next-generation computing infrastructures being developed across the world. In support of this requirement, resource allocation must support pair-wise authentication and fine-grained capabilities to transfer privilege from one administrative authority to another. For example, one domain may grant access to several of its machines to another for a week in exchange for reciprocal access rights (appropriately signed and endorsed) at some later time. In general, sites may donate to GENI as much or as little resources as they wish, with the understanding that the value of the resources that their users are granted in return may depend on the value of the resources they donate. Additionally, users or resource allocators that have been granted resource privileges should be able to transfer those privileges to other users or resource allocators, with the caveat that the individual resource owners ultimately retain the autonomy to deny access in exceptional situations.

- *Providing proper incentives.* We wish to enable resource allocation policies that: i) ensure user commitment to the continued maintenance and growth of the infrastructure and ii) promote user consumption of only as much resources as they require, especially during times of peak demand. One significant difficulty with achieving this goal is that end users will likely not (at least initially) be required to commit real money for the use of shared GENI resources. Similarly, the computation, storage, and communication resources will initially be deployed and maintained centrally. However, to achieve its ultimate goals, GENI must be self-sustaining. That is, companies and individuals must be incentivized to contribute resources to the system because they perceive a benefit from doing so.
- While developing a resource allocation system that optimally meets all of these goals is a significant research challenge, we believe that the simple two-policy model we described earlier—allowing users to request some combination of best-effort and guaranteed-share access to resources as mediated by a Science Board—will adequately meet the community's needs for the short- and medium-term.

5.2 Existing Resource Allocation Models

One important question is whether existing models for resource allocation in shared computing infrastructures would be appropriate for GENI. If so, they should be adopted outright. We find that while there are certainly significant amounts to learn, the usage model in GENI is sufficiently different from existing infrastructures, that we do require new mechanisms and policies.

Perhaps the most closely related model of resource allocation in scheduling in Supercomputer Centers. Here, hundreds to thousands of independent users share very large-scale computing infrastructures consisting of thousands of processors. While there is some variation, resource allocation proceeds around a batch scheduler. Users submit jobs to a queue. When a job reaches the front of the queue, it obtains dedicated access to the computing resources for the duration of its run. Various schedulers enable space sharing in the case where multiple jobs can run simultaneously and independently on subsets of available processors. Time sharing is rarely implemented. Administrators can set a variety of knobs to give preferential treatment to individual users, for instance, by moving particular user jobs ahead in the queue. A similar sharing model is used for accessing some shared testbeds such as Emulab and Deter.

Relative to this model, GENI's application and service mix will be fundamentally different. For instance, there will be services that will run indefinitely, making dedicated access to physical resources impractical. Further, there will be many jobs that require fine-grained resource guarantees, for instance the ability to send a train of network packets every 100ms. Finally, many applications would benefit from interactive deployment and debugging, making the unbounded time associated with waiting in a work queue impractical. Even with these limitations, we still believe that there is an important place for batch queues in GENI and hence our resource allocation mechanism should be general to batch queues, for instance, for dedicated, limited duration access to some subset of global system resources.

Also close to GENI's requirements for resource allocation are recent Grid deployments. Here, a number of sites typically federate their computing resources to run jobs that could otherwise not be supported at any individual site. In most cases however, these deployments still assume dedicated space sharing and batch queues and hence are not general to GENI's expected workload.

Finally, there have been a number of recent efforts to use auctions to allocate resources among competing users. The potential benefit of this approach is that it allows individual users to identify the relative value (their "bid") that they assign to a particular job. In this way, the system can maximize delivered utility by running those jobs that are likely to deliver the most value to their users. Other schemes typically assign fixed priorities to individual users, assigning equal weights to *all* jobs from that user (even those that may have low relative value). While such economic approaches hold some promise, a significant difficulty is that in compute environments, including GENI, they typically deal in *virtual* currency. That is, currency that may be employed only for accessing resources at a particular facility and not for any other purpose. There is then a need to invent a monetary policy, for example, policies for replenishing currency, taxing unspent currency, etc. There is also a user incentive to spend down allocated currency (or to transfer it to other users for "pennies on the dollar") even in the case where there is no real value for the associated jobs. Thus, while significant promise remains with this approach and while GENI's resource allocation mechanism should be general to any future monetary-based policy, we will not initially support auctions for GENI resource allocation initially.

5.3 Proposed Resource Allocation Mechanism

5.3.1 Capability-based Resource Transfer

Any mechanism for resource allocation should be agnostic to the specific policies that it can support. The underlying principle behind our approach is that individual domains maintain local autonomy over their local resources. This control begins with individual *Site Managers*, who may decide to grant control over a subset of locally available resources to one or more *Resource Brokers.* Initially, there will be a single logical resource broker (though it may be physically replicated) under the control of GMC. The transfer of control takes place using signed documents describing details of the privileges being transferred and the duration of the transfer. These documents, which can be thought of as capabilities, are expressed as (signed) Donations. For verification and non-repudiability, the Site Manager signs the transfer of resources in its private key (we assume some mechanism for assigning keys to individual GENI participants and will leverage the GENI security infrastructure to deliver this functionality). We omit details for dealing with malicious Site Managers that may over-promise resources or not honor future requests. The general strategy for dealing with such sites are social: Future resource requests may not be honored once it is determined that a particular site is behaving maliciously.

Expressed in XML, the Donation is written as

```
<xsd:complexType name="Donation">
    <xsd:sequence>
      <xsd:element name="GUID" type="xsd:string"/>
      <xsd:element name="Recipient" type="xsd:string"/>
      <xsd:element name="RSpec" type="tns:RSpec">
      <xsd:element name="Signature" type="xsd:base64Binary"/>
    </xsd:sequence>
</xsd:complexType>
```

The *GUID* is set by the issuing Site Manager, e.g. as the concatenation of the Site Manager's name and a sequence number of Donations it has issued. The *Recipient* is the name of the Resource Broker to whom the Donation is issued. The bulk of the Donation is encapsulated in the *RSpec*, which describes the specific resources being transferred. Expressed in XML, the RSpec is written as:

```
<xsd:complexType name="RSpec">
    <xsd:sequence>
      <xsd:element name="Issuer" type="xsd:string"/>
      <xsd:element name="ResourceGroup" type="tns:ResourceGroup"/>
      <xsd:element name="IsolationPolicy" minOccurs="0"
type="tns:IsolationPolicy"/>
      <xsd:element name="AUP" minOccurs="0" type="tns:AUP"/>
      <xsd:element name="ValidStart" type="xsd:dateTime"/>
      <xsd:element name="ValidEnd" type="xsd:dateTime"/>
    </xsd:sequence>
</xsd:complexType>
```

The *Issuer* is the name of the Site Manager issuing the RSpec. The *ResourceGroup* type is one of the standard GMC Types; it describes a set of processing, communication, measurement, and storage resources corresponding to one addressable component (e.g. a node or router). We assume that the ResourceGroup can specify fractional shares of resources, e.g., a site might donate 50% of the CPU time and 50% of the network bandwidth of a node with a CPU and network link of a particular performance, while keeping the remaining portion of the node for its own use. We further assume that the ResourceGroup specifies the sharing policy(s) (best-effort proportional-share, reservation-based guaranteed shares, etc.) supported by the component. See Section 5.5 for more details on this datatype. The optional *IsolationPolicy* describes how the component isolates users from one another and from other entities on the Internet; we currently define it as a String but expect it to become a more structured type as formal isolation policies are developed. The optional *AUP* is a String that describes the component's Acceptable Use Policy. *ValidStart* and *ValidEnd* describe the time period for which the Site Manager is transferring control over the specified resources to the Resource Broker.

In exchange for granting these capabilities to a Resource Broker, the Site Manager in turn receives a broker-signed capability granting access to some portion of global system resources. While the resources granted by the Site Manager are specific (e.g., individual machines for some period of time), the resources received in return are abstract. The value of these credits strongly depends upon the specific resource allocation policy that GENI wishes to enforce. For example, the centralized GENI resource broker may hand out "unit" credits to all GENI sites. In this case, the capabilities may allow all sites to instantiate slices on all GENI nodes and to compete for resources in a best-effort manner. Alternatively, the credits may be proportional to the perceived value of the resources being allocated to GENI such that sites contributing more resources may receive a higher proportional weight when competing for global system resources.

We call these credits Tokens and define them as follows:

```
<xsd:complexType name="Token">
    <xsd:sequence>
      <xsd:element name="Issuer" type="xsd:string"/>
      <xsd:element name="GUID" type="xsd:string"/>
      <xsd:element name="Recipient" type="tns:SliceName"/>
      <xsd:element name="Value" type="xsd:decimal"/>
      <xsd:element name="ValidStart" type="xsd:dateTime"/>
      <xsd:element name="ValidEnd" type="xsd:dateTime"/>
      <xsd:element name="ParentGUID" type="xsd:string"/>
      <xsd:element name="Signature" type="xsd:base64Binary"/>
    </xsd:sequence>
</xsd:complexType>
```

Tokens are akin to resource claims in SHARP. The Value field indicates the abstract value of the Token. This value is interpreted later by the Resource Broker when a user asks the Resource Broker to convert a Token into a lease on concrete resources (a Ticket), as described in the next section. We intentionally do not constrain the interpretation of this field to allow flexibility for future work in resource allocation policies. For example, one Resource Broker policy might specify that a Token of value 1 may be redeemed for best-effort access on any single node, and that Tokens of value greater than 1 may be redeemed for some guaranteed share. A different policy might auction a 100% share of a component to the highest bidder. The *ParentGUID* field of the Token allows tokens to be chained together and subdivided for delegation. The *ValidStart* and *ValidEnd* fields of the Token would generally correspond to the time period for which the site has offered its resources in return for the Token.

Once a site, via its Site Manager, has offered resources to a Resource Broker and the Resource Broker has returned one or more Tokens, the Site Manager can subdivide those tokens among the site's users. For example, a Site Manager that has been granted a token of value 1000 might issue tokens of value 100 to 10 of the site's users. Each of these 10 tokens would be chained to the original token to allow detection of Site Managers that issue more total token value than they are entitled to issue. In general, users or Site Managers could receive tokens from a Site Manager, another user, or the Science Board, a GENI-administered entity that allocates additional Tokens to users or sites that are deemed to offer particularly valuable systems or experiments. Users or sites may subdivide tokens for their own use (e.g., dividing a token of value 100 into 100 separate tokens each of value 1 to enable proportional-share access to 100 separate nodes) or to share with other users or other sites. Whether such credit-pooling is allowed is a question of GMC policy. For instance, it is easy for GMC to disallow the use of any credits that have been transferred from one user to another, because the software verifying the validity of a given capability verifies the entire chain of transfers.

The steps we have described thus far can be thought of as a “background” process by which sites periodically renew their offer of resources to the system and keep their users supplied with abstract credits (Tokens). Next we describe how a user locates resources and redeems Tokens for concrete leases on specific resources that they wish to use.

5.3.2 Locating Resources and Redeeming Tokens for Tickets

Once a user has received Tokens, the user leverages a resource discovery mechanism to find an appropriate set of resources to host their experiment or service; see Section 6 for a detailed discussion of the resource discovery process. We intentionally do not constrain the resource discovery mechanism in our discussion of resource allocation, as we expect GENI to host many different resource discovery services..

Once the user has identified a component they wish to use, the user presents the Resource Broker (to whom the component's Site Manager has granted control over some or all of the component's resources) with a Token and a description of the desired component usage. This description may be, but is not required to be, expressed as an RSpec. The Resource Broker verifies the validity of the Token, possibly checks for authorization with the Component Manager on the component the user wishes to access, and returns a Ticket. A Ticket is a lease – it grants to a slice access to the resources specified in an embedded RSpec for a specified duration of time:

```
<xsd:complexType name="Ticket">
      <xsd:element name="GUID" type="xsd:string"/>
      <xsd:element name="Recipient" type="tns:SliceName"/>
      <xsd:element name="RSpec" type="tns:RSpec"/>
      <xsd:element name="ValidFor" type="xsd:duration"/>
      <xsd:element name="Signature" type="xsd:base64Binary"/>
</xsd:complexType>
```

The Ticket is a promise by a Resource Broker that the component will offer the resources specified by the Ticket's RSpec to the *Recipient* for up to *ValidFor* time units starting any time at or after the RSPec's*ValidStart*, but not to extend past the RSPec's *ValidEnd*. The *GUID* is set by the issuing Resource Broker, e.g. as the concatenation of the Broker's name and a sequence number of Tickets it has issued. The *Recipient* is a slice name rather than a Resource Broker as in a Donation. The other fields of the RSpec are interpreted as described in Section 5.3.1, but now the RSpec represents a promise from a Resource Broker to a user rather than a promise from a Site Manager to a Resource Broker. We assume that the RSpec's ResourceGroup specifies the nature of the resource guarantee, e.g., whether it is best-effort proportional-share access or a guaranteed minimum fixed share. A Ticket may specify a time period arbitrarily far in the future, which equates to a guaranteed "reservation" for the user at that future time.

The Resource Broker may refuse to honor a Ticket for any number of reasons. For example, a Ticket may be refused if the user makes a guaranteed-share request and there are insufficient reservable resources available during the desired interval. Likewise, to prevent thrashing under a proportional-share policy, a Ticket may be refused if the user makes a proportional-share request for a resource that the Resource Broker or Component Manager deems overcommitted. A future extension to our described mechanism might allow the Resource Broker to queue such requests and issue a callback to the requester when the desired resource becomes available. For now, however, we assume that such queueing is handled by an out-of-band exchange (between the user and Resource Broker) that is invoked when a request is denied, and that the user resubmits the denied request later if they still wish to use the resource.

Finally, some time between the *ValidStart* and *ValidEnd* times on the Ticket's RSpec, the user presents the Ticket to the Component Manager responsible for the component the user wishes to access. The Component Manager creates a Sliver on the component and returns it to the user. A Sliver is a standard GMC Type and is described elsewhere.

Our goal is thus to allow Tokens to express a wide range of policies as GENI evolves. Initially, we expect each user to be given a token of sufficient value that they can subdivide it into enough smaller-value Tokens to access every machine in GENI, though a single PropShare token may be defined to be infinitely divisible. Later, Tokens can be used to encode more complex resource access privileges. For instance, if one system node is deemed particularly valuable, Resource Brokers might implement policies that require a token of value greater than 1 if the user wishes a ticket for that resource, or the Resource Broker might only offer a Ticket of short duration in return for a token of value 1. Analogously, the Science Board might deem a particular user's work as particularly valuable, and issue them extra Tokens. The user could then select some appropriate set of resources to "purchase" for their experiment or service through the GMC-run resource broker. Some combination of the Science Board and the resource broker would be responsible for replenishing currency to individual sites and users, as well as for updating per resource-hour pricing. Also, the Science Board may create special Tickets that grant privileges to access a given component, bypassing the Token scheme altogether.

Note that the data structures we have described all have limited lifetimes. In a deployed system, Site Managers are continually renewing their offer of resoures to Resource Brokers, who are continually issuing new Tokens in return, which the Site Managers divide up among users, who in turn redeem those Tokens to obtain or renew Tickets that are used to obtain or renew slivers.

To ensure that users eventually become aware of all donated resources, we encourage the resource discovery services to populate their resource databases using Resource Broker data (possibly augmented with additional information provided by the components themselves) to ensure that all resources offered to the system by Site Managers are advertised as available to GENI users. In particular, it is not safe to trust Site Managers to advertise the resources they have promised to a Resource Broker in return for Tokens; they could offer resources to a Resource Broker in order to obtain Tokens, but then never advertise the offered resources, making those resources unknown to users. A Resource Broker can prevent this abuse by providing the RSpecs it has received to all known resource discovery services, and/or by periodically probing known resource discovery services to ensure that promised resources are in fact being advertised as available to system users.

An extension to the Resource Broker would provide transactional access to multiple resources. Because a single centralized resource broker mediates access to global GENI resources, it is also the entity in the position to coordinate simultaneous access to multiple resources. In this model, a user may make a request to the Resource broker Specifying *all-or-nothing* semantics, i.e., either the user obtains Tickets (valid for overlapping time periods) for all the specified resources or none at all. Thus, it is the task of the resource broker to track resource availability on a per-resource basis and to ensure that individual resources are not overbooked.

Sections 5.4 recaps the datatypes presented above, and details the messages passed among the User, Resource Broker, Site Manager, and Component Manager to request, renew, and receive Tokens, Tickets, and Slivers.

5.3.3 Acceptable Use Policies

One interesting question is the inter-play between Acceptable Use Policies (AUPs) at individual sites and the GENI-wide AUP. It could be the case that individual site AUPs will be more restrictive than the GENI AUP. For example, one company may decide that it does not wish to run jobs on behalf of certain other companies. We leave the question of whether individual site AUPs should be enforced to a GENI policy decision but there are interesting tradeoffs. Allowing sites to express AUPs (and having the necessary mechanisms to enforce per-site AUPs) will likely broaden the appeal of GENI, allowing more sites to join. At the same time, it will likely add significant system-wide complexity. Further, if sites are given the option of imposing restrictions on local resource usage, they likely will take advantage of it, fragmenting the utility of global resources. One mechanism to push back on this desire is to reduce the number of credits granted to sites that wish to impose local AUPs. In one model, sites wishing to impose any restrictions would receive no credits (meaning that GENI in effect disallows per-site AUPs). At the other extreme, the value of individual site resources in combination with their AUP would have to be negotiated on a per-site basis. Regardless, if per-site AUPs are to be allowed, then some mechanism would have to be designed to both specify and enforce AUPs.

5.3.4 Resource Allocation Policies

A variety of interesting policies become possible with the above mechanism. For instance, GENI Resource Brokers may hold credits corresponding to some fraction of global resources back for particular users/sites that run services judged to be of particular value to either the GENI or the external community. Similarly, socialist policies become possible where GENI may allocate resource privileges to sites that are not directly contributing resources or to allocate more Tokens than would normally be available in exchange for a resource pool.

Site managers may allocate available Tokens to a user based on the judged priority of the work being performed by that user. An available management interface should make it straightforward for the site manager to re-allocate available credits among users as priorities change (changes would take effect when the Tickets associated with an application expire, at which point the user's application would need to obtain new Tickets from the Resource Broker).

Initially, GENI will use a very simple policy enabled by the mechanisms we have described. In particular, all sites will be required, by contract, to donate control over all of their resources to the single logically centralized Resource Broker. The Resource Broker will in return send Tokens to sites based on the Science Board's judgment of that site's science priority; it may also grant additional Tokens to each site for educational purposes. In this case, the "exchange rate" between donated resources and Tokens is manually configured in the Resource Broker on a site-by-site basis. As additional Resource Brokers are added to the system during later stages of the project, they will be able to set their own "exchange rate" but the Science Board will still be able to directly grant Tokens and Tickets to sites and users in recognition of the science value of their projects and resources.

Note that the mechanisms we have described allows both on-demand and batched access to experimental infrastructure. The latter is desirable for resources to which users want dedicated access, such as a shared wireless testbed. In this case, the Resource Broker allocates a Ticket whose *ValidFor* is smaller than the time between the Ticket's RSpec's *ValidStart* and *ValidEnd*. The user could then present this Ticket to a batch scheduler granting access to the resource who would perform admission control among appropriately privileged users, giving this user dedicated access for *ValidFor* time units or until the job completes (in a model similar to the user of DETER of Emulab for instance). This type of batch scheduling is also useful for handling reservations for guaranteed-share access to resources.

5.3.5 Support for Decentralization and Federation

In the future, it should also be possible to extend centralized GENI (GMC) resource brokers to enable federation. For example, sites may decide to interact with a variety of third-party resource brokers (implementing different exchange rates, levels of security, control over different resource subsets, etc.). Importantly, the resource allocation mechanism proposed in this section must be compatible with any future decentralization. That is, it should be a matter of policy to move to a more decentralized or federated resource allocation system; such a move should not require rearchitecting the underlying capability-based mechanism. For instance, using the existing mechanism, sites may decide to forego going through resource brokers altogether by entering into pair-wise peering agreements, much like ISP's do today for bandwidth. That is, sites could run their own Resource Brokers, issuing Tokens to one another to be redeemed for access to a particular set of local resources in exchange for access to some set of remote resources. Delegated Tokens can then form the basis of a decentralized model for access to global resources based only on pair-wise peering agreements [18].

One primary benefit of this approach is support for federation. Additional international compute infrastructures can use this flexible capability scheme to enter into agreements describing the exact set of resources to be exchanged among independent organizations. Consider the case where GENI wishes to federate with some similar international testbed. GMC would then be responsible for negotiating the access rights to the remote infrastructure, e.g., GENI grants access to 100 abstract machines (including CPU, memory, storage, and network) for the next week in exchange for Tokens redeemable for a like amount of resources in the remote testbed. As long as the two organizations agree to the mechanism for specifying resource privileges at the boundary of their organizations, they are free to implement arbitrary mechanisms for encoding resource privileges, performing resource discovery, embedding slices, etc. within their local organization. This follows with analogy to BGP (an agreed upon protocol between organizations) that allows organizations to run whatever routing mechanism / policy they wish within their organization. Thus, one hope for the capability-based resource allocation mechanism described in this section is that it is general enough to become the mechanism supporting both intra- and inter-site resource allocation (though it should not require adoption within a site, just between cooperating sites).

One interesting side effect of supporting federation with multiple resource brokers is that it becomes more difficult to support transactional access to resource sets controlled by multiple independent brokers. To continue to support federation, the resource brokers could be extended to support some “tentative resource commit” in support of a distributed two-phase commit protocol to support transactional access across multiple resource brokers. Alternatively, one Resource Broker (A) might transfer control over some of its resources to another Resource Broker (B) using the Donation mechanism normally used for sites to donate resources to a Resource Broker. Then a user wishing to access resources originally controlled by A and B could interact just with broker A, eliminating the need for a distributed commit protocol between Resource Brokers.

5.4 Required Software Components

The above discussion suggests the following set of software components to support the target resource allocation mechanisms:

- *Science Board Interface.* The Science Board requires some mechanism for granting Tokens to users.

- *Resource Broker.* The resource broker is responsible for granting Tokens to individual site managers that offer Donations of resources to GENI, and also for issuing Tickets to users for requested resources. Resource Brokers must validate end user identities and verify their resource privileges according to GENI policies.

- *Site Manager.* PIs at individual sites require an interface for enabling end user accounts and assigning Tokens to user slices upon request.

- *Component Manager.* End users will present Tickets obtained from Resource Brokers to be bound to individual virtual machines/slivers. The Component Manager is responsible for instantiating the virtual machine and binding the appropriate set of resources to it.

- *End User Interface.* End users require an interface for managing their available resource privileges (handed to them by Site Managers and the Science Board). In the simplest realization, the end user interface is part of the Site Manager, which stores these privileges on behalf of all users/slices affiliated with that site, indexed by public keys. In a more flexible architecture, end users must store and maintain these privileges themselves. End users require a *resource discovery* mechanism to find resources appropriate to host their computation and a *programmatic interface* to the Resource Broker to turn concrete resource requests into Tickets, and to the Component Managers to turn Tickets into individual slivers/virtual machines that collectively will make up the user’s slice of GENI resources.

In the remainder of this section we provide XSD definitions of the datatypes and messages that implement the resource allocation mechanisms we have described. These definitions augment the existing GMC datatypes and messages described at http://www.geni.net/wsdl.php, except where we have specifically noted we are modifying those types and/or messages.

5.4.1 Resource Allocation Protocol Datatypes

Most of these datatypes are described in Section 5.3.1 and Section 5.3.2; we repeat them here for completeness. Note that additional datatpyes (some referenced by the below datatypes) are defined at
http://www.isi.edu/~faber/GMC/schemas/resources.xsd

The **RSpec** represents resources donated from a Site Manager to a Resource Broker (when it is encapsulated in a **Donation**), or a Resource Broker's guarantee to a user for access to the specified resources (when it is encapsulated in a **Ticket**).

```
    <xsd:complexType name="RSpec">
      <xsd:sequence>
        <xsd:element name="Issuer" type="xsd:string"/>
        <xsd:element name="ResourceGroup" type="tns:ResourceGroup"/>
        <xsd:element name="IsolationPolicy" minOccurs="0"
  type="tns:IsolationPolicy"/>
        <xsd:element name="AUP" minOccurs="0" type="tns:AUP"/>
        <!--xsd:element name="RequiredTokens" type="xsd:decimal"/-->
        <xsd:element name="ValidStart" type="xsd:dateTime"/>
        <xsd:element name="ValidEnd" type="xsd:dateTime"/>
      </xsd:sequence>
    </xsd:complexType>
```

A **Donation** is a handing of control of resources from a Site Manager to a Resource Broker. The **IsolationPolicy** and **AUP** describe the policy the resource uses to isolate users from one another, and the resource's Acceptable Use Policy, respectively.

```
    <xsd:complexType name="Donation">
      <xsd:sequence>
        <xsd:element name="GUID" type="xsd:string"/>
        <xsd:element name="Recipient" type="xsd:string"/>
        <xsd:element name="RSpec" type="tns:RSpec">
        <xsd:element name="Signature" type="xsd:base64Binary"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:simpleType name="IsolationPlicy">
      <xsd:list itemType="xsd:string"/>
    </xsd:simpleType>

    <xsd:simpleType name="AUP">
      <xsd:list itemType="xsd:string"/>
    </xsd:simpleType>
```

A **Token** is an abstract capability, or credit, that a Site Manager receives from a Resource Broker in return for a resource Donation, and that a user receives from their Site Manager according to the site-specific policy. It is presented to a Resource Broker, along with a resource request, to obtain a lease on concrete resources (a **Ticket**). We also introduce the **TokenSet** to allow multiple Tokens to be passed as a group. A **TokenRequest** is a request from a user to their Site Manager for tokens.

```
    <xsd:complexType name="Token">
      <xsd:sequence>
        <xsd:element name="Issuer" type="xsd:string"/>
        <xsd:element name="GUID" type="xsd:string"/>
        <xsd:element name="Recipient" type="tns:SliceName"/>
        <xsd:element name="Value" type="xsd:decimal"/>
        <xsd:element name="ValidStart" type="xsd:dateTime"/>
        <xsd:element name="ValidEnd" type="xsd:dateTime"/>
        <xsd:element name="ParentGUID" type="xsd:string"/>
        <xsd:element name="Signature" type="xsd:base64Binary"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="TokenSet">
      <xsd:element name="Token" minOccurs="1" maxOccurs="unbounded"
  type="tns:Token"/>
    </complexType>

    <xsd:complexType name="TokenRequest">
      <xsd:sequence>
        <xsd:element name="RequesterSlice" type="tns:SliceName"/>
        <xsd:element name="RequesterUser" type="tns:UserInfo"/>
        <xsd:element name="GUID" type="xsd:string"/>
        <xsd:element name="Value" type="xsd:decimal"/>
      </xsd:sequence>
    </xsd:complexType>
```

A **Ticket** is a lease on concrete resources, which are described by the embedded **RSpec**.

```
    <xsd:complexType name="Ticket">
      <xsd:sequence>
        <xsd:element name="GUID" type="xsd:string"/>
        <xsd:element name="Recipient" type="tns:SliceName"/>
        <xsd:element name="RSpec" type="tns:RSpec"/>
        <xsd:element name="ValidFor" type="xsd:duration"/>
        <xsd:element name="Signature" type="xsd:base64Binary"/>
      </xsd:sequence>
    </xsd:complexType>
```

5.4.2 Messages for Component Manager service

(These operations are slight modifications to the existing CreateSliver and ModifySliver messages in the GMC Specification, to incorporate the new datatypes defined earlier. Comments from original are left intact.)

```
  <definitions name="Component_Manager"
    targetNamespace="http://tg4.ucsd.edu/~davidopp/CMmessages"
    xmlns:tns="http://tg4.ucsd.edu/~davidopp/CMmessages"
    xmlns:xsd1="http://tg4.ucsd.edu/~davidopp/GMC_types"
    xmlns:xsd2="http://tg4.ucsd.edu/~davidopp/RA_types"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Generic error message

```
    <message name="GMCErrorMessage">
      <part name="GMCError" type="xsd1:GMCError"/>
    </message>
```

Messages between user to Component Manager to create and modify slivers

```
    <message name="CreateSliverRequestMessage">
      <part name="UserInfo" type="xsd1:UserInfo"/>
      <part name="Ticket" type="xsd2:Ticket"/>
    </message>

    <message name="CreateSliverResponseMessage">
      <part name="Sliver" type="xsd1:Sliver"/>
    </message>

    <message name="CreateSliverErrorMessage">
      <part name="GMCError" type="xsd1:GMCError"/>
    </message>

    <message name="ModifySliverRequestMessage">
      <part name="UserInfo" type="xsd1:UserInfo"/>
      <part name="SliverName" type="xsd1:SliverName"/>
      <part name="Ticket" type="xsd2:Ticket"/>
    </message>

    <message name="ModifySliverResponseMessage">
      <part name="Sliver" type="xsd1:Sliver"/>
    </message>

    <message name="ModifySliverErrorMessage">
      <part name="GMCError" type="xsd1:GMCError"/>
    </message>
```

[additional messages are defined at
http://www.isi.edu/~faber/GMC/messages/CMmessages.wsdl]

Operations from user to Component Manager to create and modify slivers

```
    <portType name="CMPortType">

      <operation name="CreateSliver">
        <documentation>
          Realize a ticket - that is, allocate resources on the
  component to a
```

```
        valid slice on behalf of a user.  The ticket name and user
credentials
        are parameters and the result is a description of the sliver
actually
        allocated.
      </documentation>
      <input message="tns:CreateSliverRequestMessage"/>
      <output message="tns:CreateSliverResponseMessage"/>
      <fault name="CreateSliverFault"
message="tns:CreateSliverErrorMessage"/>
    </operation>

    <operation name="ModifySliver">
      <documentation>
        Change the resource allocation of a sliver on this component
or renew
        the sliver. There will be a notification to the sliver if
this succeeds.
        User credentials and sliver validity are checked and if the
allocation is
        successful a new sliver is returned.  The sliver will have
the same
        name as the one passed in.  We return the full sliver so
that partial
        resource allocations may be permitted.  A caller should
confirm that
        the returned sliver contains the resources that the caller
requested.
      </documentation>
      <input message="tns:ModifySliverRequestMessage"/>
      <output message="tns:ModifySliverResponseMessage"/>
      <fault name="ModifySliverFault"
message="tns:ModifySliverErrorMessage"/>
    </operation>
```

[additional operations are defined at http://www.isi.edu/~faber/GMC/messages/CMmessages.wsdl]

```
  </portType>
</definitions>
```

5.4.3 Messages for Resource Broker service

```
<definitions name="ResourceBroker"
  targetNamespace="http://tg4.ucsd.edu/~davidopp/RBmessages"
  xmlns:tns="http://tg4.ucsd.edu/~davidopp/RBmessages"
  xmlns:xsd1="http://tg4.ucsd.edu/~davidopp/GMC_types"
  xmlns:xsd2="http://tg4.ucsd.edu/~davidopp/RA_types"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Messages between Site Manager to Resource Broker to request tokens in return for Donation

```
<message name="SMGetTokensRequestMessage">
  <part name="Donation" type="xsd2:Donation"/>
</message>

<message name="SMGetTokensResponseMessage">
  <part name="TokenSet" type="xsd2:TokenSet"/>
</message>

<message name="SMGetTokensErrorMessage">
  <part name="GMCError" type="xsd1:GMCError"/>
</message>
```

Messages between user and Resource Broker to request and modify Tickets

```
<message name="GetTicketRequestMessage">
  <part name="User" type="xsd1:UserInfo"/>
  <part name="Ticket" type="xsd2:Ticket"/>
  <part name="Token" type="xsd2:Token"/>
</message>

<message name="GetTicketResponseMessage">
  <part name="Ticket" type="xsd2:Ticket"/>
</message>

<message name="GetTicketErrorMessage">
  <part name="GMCError" type="xsd1:GMCError"/>
</message>

<message name="ModifyTicketRequestMessage">
  <part name="User" type="xsd1:UserInfo"/>
  <part name="OldTicketGUID" type="xsd:string"/>
  <part name="NewTicket" type="xsd2:Ticket"/>
  <part name="Token" type="xsd2:Token"/>
</message>

<message name="ModifyTicketResponseMessage">
  <part name="Ticket" type="xsd2:Ticket"/>
</message>

<message name="ModifyTicketErrorMessage">
  <part name="GMCError" type="xsd1:GMCError"/>
</message>
```

Operations to grant Tokens to sites and Tickets to users

```
<portType name="GMCRBPortType">
  <operation name="GrantTokensToSite">
    <documentation>
```

```
        Receive a Donation from a Site Manager, return a set of
tokens
        (one of each type required) each of which
        will be divided up by the Site Manager to give out to
slices.
      </documentation>
      <input message="tns:SMGetTokensRequestMessage"/>
      <output message="tns:SMGetTokensResponseMessage"/>
      <fault name="SMGetTokensFault"
message="tns:SMGetTokensErrorMessage"/>
    </operation>

    <operation name="GetTicket">
      <documentation>
        Receive a request for a Ticket from a user. The RB may
forward this
        to the Component Manager for authorization before issuing
the Ticket.
      </documentation>
      <input message="tns:GetTicketRequestMessage"/>
      <output message="tns:GetTicketResponseMessage"/>
      <fault name="GetTicketFault"
message="tns:GetTicketErrorMessage"/>
    </operation>

    <operation name="ModifyTicket">
      <documentation>
        Receive a request from a user to renew or modify a Ticket.
      </documentation>
      <input message="tns:ModifyTicketRequestMessage"/>
      <output message="tns:ModifyTicketResponseMessage"/>
      <fault name="ModifyTicketFault"
message="tns:ModifyTicketErrorMessage"/>
    </operation>


  </portType>
</definitions>
```

5.4.4 Messages for Site Manager service

```
<definitions name="SiteManager"
  targetNamespace="http://tg4.ucsd.edu/~davidopp/SMmessages"
  xmlns:tns="http://tg4.ucsd.edu/~davidopp/SMmessages"
  xmlns:xsd1="http://tg4.ucsd.edu/~davidopp/GMC_types"
  xmlns:xsd2="http://tg4.ucsd.edu/~davidopp/RA_types"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Messages between Site Manager and user who requests Tokens

```
    <message name="UGetTokenRequestMessage">
      <part name="TokenRequest" type="xsd2:TokenRequest"/>
    </message>

    <message name="UGetTokenResponseMessage">
      <part name="Token" type="xsd2:Token"/>
    </message>

    <message name="UGetTokenErrorMessage">
      <part name="GMCError" type="xsd1:GMCError"/>
    </message>

    <portType name="GMCSMPortType">
```

Operation to grant Tokens to a user in response to their request

```
      <operation name="GrantTokenToUser">
        <documentation>
          Receive a token request from the user, return it if
  authorized. Although primarily
          a service provided by the Site Manager, also provided by the
  Science Board.
        </documentation>
        <input message="tns:UGetTokenRequestMessage"/>
        <output message="tns:UGetTokenResponseMessage"/>
        <fault name="UGetTokenFault"
  message="tns:UGetTokenErrorMessage"/>

    </portType>
  </definitions>
```

5.5 Resource Descriptions for the GENI Facility

This section is a description of the resources available to experimenters in the GENI facility. The resources described here will be allocated by the methods described in this section.

5.5.1 Roles of GENI

GENI will play three fundamental roles for the networking and distributed systems research community in the United States:

1. As a *laboratory*: a facility for *controlled, repeatable, reproducible* experiments under safe conditions. This facility should provide specific, precise, guaranteed conditions for the conduct of experiments, and mutual protective guarantees for facility users and third parties.

2. As an *observatory*: a facility for precise, non-invasive observations of the behavior of existing networks and distributed systems under current network conditions.

3. As a *field experimental station* where new systems can be tested under actual network conditions.

These three classes of usage have slightly varying resource requirements, and resource descriptions must fit all of them. Moreover, the resource descriptions must dexfribe items that are realizable with available technology in the GENI timeframe.

There are three systems on whose experiences we draw, as representatives of the GENI frameworks: Emulab/Netbed, PlanetLab, and Tycoon.

Emulab, from the University of Utah, is the premiere distributed systems and networking laboratory facility. It offers a controlled testing environment consisting of bare machines, with images loadable from a centralized resource, and private links with a maximum bandwidth of 100 Mb/sec and controllable impairments.

PlanetLab, centered at Princeton, is the leading distributed systems and networking observatory and field experimentation station. It is an overlay network of bare virtualizable IA-32 Linux machines, with virtualization at the syscall level using Vservers, a Linux equivalent of BSD jails. Connectivity is over the open Internet, using physical links and bare IPv4 addresses. Bandwidth is capped to control expenses at the hosting sites. Only Linux executables are loadable.

Tycoon, from HP Labs, is a market-based cluster management system. Tycoon allocates virtual machines based on the Xen virtual computing base, with memory, CPU, and bandwidth set by the end-user/developer. While this is not specifically a networking testbed, its more general use as a cluster manager makes this an interesting model for experimental resource allocation: viewed as an abstract problem, network observation and experimentation is simply another cluster application: more properly, as an application for networks of clusters with strong isolation requirements.

The most important common feature of all three environments is that they offer the developer/user bare machines: in the case of PlanetLab and Tycoon, bare virtual machines, and in the case of Emulab, bare physical machines. The choice of bare machines is not accidental: conflicts in software environments are a major deployment barrier, and use of bare machines therefore is a prerequisite for rapid deployment.

Resource descriptions describe the intersection between the needs of researchers and the technical ability of the facility to offer resources. For example, a researcher may desire a line with guaranteed latency and bandwidth routed over the commodity Internet; this is beyond the scope of current technology. Moreover, resource descriptions must be unambiguous and easily matched with available resources, and should therefore be as simple as possible without imposing undue burdens on the researcher. For example, it would be a convenience to a researcher to specify that his virtual machine come preloaded with a specific software suite. The likely value of that convenience must be weighed against the probability of error in the suite description, the burden imposed on resource provider and broker to find a virtual machine with the right software environment.

5.5.2 Naming Resources

In order to be useful to GENI experimenters, allocated resources must be *named*. The names of resources used in an experiment will be encoded in programs and scripts. Further, a GENI experiment will in general be virtual and mobile: it will be remapped to different physical hardware between, and sometimes during, experimental sessions.

The constraints of mobility and names encoded in programs and scripts gives us an immediate design consideration: names in a GENI experiment last longer than invocations onto physical hardware.

There are two models of distributed names: the PlanetLab model and the Emulab model. We detail these here:

- In the PlanetLab namespace, node names are names assigned by the local administrator. The experiment name is bound as a user name (the "slice name") on each node. User names are nonexistent: they are simply referents of public keys. There is a single authentication protocol to a node.

- In the emulab namespace, node names are virtualized: they are of the form

  nodename.experimentname.projectname.emulab.net

  e.g. westc.saic4.chart.emulab.net. User names are user-selected and global across the testbed. Authentication is either by key or by password, and there are the standard set of password recovery mechanisms

Without discussing here the merits of the emulab and PlanetLab naming conventions for users, we turn to the naming of nodes and communication elements. It is clear that the emulab model has a number of advantages. All names are selected by the experimenter, with a disambiguating postfix. The naming is easily supported by standard DNS techniques. Scripts, programs, and configuration files can be written once, at the time the experiment is created or resources are added to it, not each time the experiment is mapped to specific hardware.

The second major constraint is the avoidance of collision in the namespace. This can be achieved by names allocated by a central authority, or by partitioning the namespace . In general, the namespace roots in a partitioned namespace should be chosen to permit experiments sharing a root to choose the names at the next level by social negotiation.

An interesting design choice is free choice vs allocation of namespace roots. In most Internet domains, the root level is freely chosen on a first-come first-served basis. Since there is commercial value to many names, this has led to wasted motion on allocation and contention of allocation over names.

In contrast, there seems to be no universally-preferred procedure for central allocation of names. One procedure, used in Usenet had no central namespace at all. All names were relative to a specific local namespace, and were resolved locally. The user watcom!allegro!ucbvax!jkf reached a user named jkf, by chaining through hosts watcom, then allegro, then ucbvax. All names were local, however: ucbvax only had meaning in the namespace designated allegro in the namespace designated watcom, and of course, the actual host anywhere along the chain could be changed at any time.

The Usenet protocol and namespace grew up by happenstance, but it is well-suited to GENI. In the first place, it is desirable to permit a loose-enough namespace convention to permit experimentation in this area.

Every GENI user must have a unique user id, in order to create experiments and networks. This user id must be registered with an authority that permits the user to create experiments and networks. For the moment, and for simplicity, we assume the authority geni.net.

The user's root name service is always userId.geni.net. From here, resolution proceeds strictly by reference in prefix order from the root, with names delimited by a percent sign (%). The partial reference chain …%x%y%z.. indicates that x is a name in namespace y, which is a name in namespace z.

There are several advantages of this naming scheme:

1. It does not bind experimenters to the Internet naming conventions, permitting experiments in name resolution: an experimenter simply puts in a different namespace resolver at a specific (usually terminal) point in the chain.
2. It permits users to use different namespaces for different roles. For example, and experimenter associated with two different institutions could choose different root namespaces under his userid.
3. It permits users to offer name resolution services, though the mechanism of *soft chaining,* described below

Soft Chaining

Soft chaining is a mechanism wherein users alias a namespace to a namespace hosted by another user. It is the GENI namespace equivalent of a soft link in a Unix filesystem. In this facility, the user declares to the GENI name resolution service the link:

5.5.3 Computation Environments

There is remarkable unanimity in existing shared testbed and cluster facilities in descriptions of available computational resources: bare virtual machines, with a processor architecture, processor speed, available memory, cache, and disk. We add to this specification of network interfaces. For the purposes of this document, we assume perfect resource isolation on a virtual machine, and, further, that any VMs of any size can be accommodated up to the physical machine limits in the testbed. The maximum possible values of all parameters will of course increase over the lifetime of the testbed.

A specific, subtle issue is in connectivity specification, since this forms a major part of the contract between the virtual machine provider and the developer and experimenter. Indeed, it is reasonable to argue that most anticipated GENI security issues revolve around the issue of connectivity. The issue of most anticipated concern is *external* connectivity: connectivity of testbed machines to hosts outside the testbed. Of course, external connectivity of some sort is convenient for purposes of experimental control and monitoring. It is also integral to some of GENI's stated purposes. Both experiments that monitor the external Internet (such as the Array of Telescopes experiment from UC-Berkeley and UCSD, or various experiments which monitor peer-to-peer traffic) and services offered to end-users (such as end-system multicast and content distribution networks such as CoDeeN and Coral) must connect outside of the testbed.

There are two central issues. First, ensuring that the experiment complies with AUPs at the hosting site, and, second, that the experimenter restrict connectivity and external connection activity to that required for the experiment. This issue intersects with resource descriptions in that the central role of a resource broker is to match demands and resources. The resources always come with strings; it's therefore important to make the strings visible.

For the moment, we assume a set of discrete, enumerated, AUPs which sites register with GENI. In a virtual machine request, the experimenter enumerates the set of AUPs to which he asserts his experiments comply, and the resource broker considers only those sites who have registered those AUPs.

Connectivity is expressed as a set of configurable firewalls on interfaces. From where will this machine accept connections? To where will it attempt to make connections? Which ports should permit incoming/outgoing connections?

In general, the default on an interface is that all connections are accepted, and all can be made. Restrictions are placed through the use of *firewalls*. Ultimately, a firewall specifies a set of identifiers (port lists and IP addresses) and restrictions: accept or send, permit or deny. IP addresses can be symbolically addressed: for example, EXPERIMENT_ONLY. And the complement of PORTs/IP_ADDRESSES are specified by specifying that this is a list of Ports/IP Addresses EXCLUDED from this general rule.

This leads to a solvable constraint system on the interface.

```
<VIRTUAL_MACHINE>
     <PROCESSOR Architecture={IA32, IA64, Itanium, PowerPC, Sparc,
or other}
       MIN_SPEED=Unspecified | Real Value in GHz
       MAX_SPEED=Unspecified | Real Value in GHz
       MIN_CACHE=Unspecified | Integer Value in Kbytes
       MAX_CACHE=Unspecified | Integer Value in Kbytes/>
    <MEMORY MIN_SPEED=Unspecified | Real Value in MHz
       MAX_SPEED=Unspecified | Real Value in MHz
       MIN_SIZE=Unspecified | Integer Value in Mbytes
       MAX_SIZE=Unspecified | Integer Value in Mbytes/>
   <DISK  MIN_SPEED=Unspecified | Integer Value in RPM
       MAX_SPEED=Unspecified | Integer Value in RPM
       MIN_SIZE=Unspecified | Integer Value in Gbytes
       MAX_SIZE=Unspecified | Integer Value in Gbytes/>
   <AUP = {list of complied AUPs}/>
    <NIC  Number = 0 to N>
          <INTERFACE NAME=URL (OPTIONAL) MIN_SPEED=Integer Value in
Mbs
                                  MAX_SPEED= Integer Value in Mbs>
              <FIREWALL PORTS={LIST | ALL}
                         PORT_INCLUSION = (EXCLUDE|INCLUDE)
                         RESTRICTION={ACCEPT|SEND}
                         IP_ADDRESSES = {IP_ADDRESS_LIST |
                    EXPERIMENT_ONLY}
                         ADDRESS_INCLUSION = (INCLUDE|EXCLUDE)
                         VALUE=(PERMIT|DENY) />

               <FIREWALL…/>
           </INTERFACE>
           <INTERFACE MIN_SPEED=Integer Value in Mbs
                                  MAX_SPEED= Integer Value in Mbs/>
             …(to N)
     </NIC>
   </VIRTUAL_MACHINE>
```

Architecture types should be specified as a string, since they are likely to change over the life of the testbed. However, allowable types should be enumerated at the facility, with a short English-readable description of the architecture. Emulab offers an excellent example of specific machine types.

It is important to note here that virtualized resources must be guaranteed once acquired. In particular, the physical resource underlying a collection of virtual machines should have enough physical resources to support the sum of the demanded virtual resources.

It should be noted here that the Virtual Machine is the equivalent of a PlanetLab sliver.

5.5.4 Clusters

It has been observed that some distributed systems require clusters of machines to perform their task. One famous example is the OceanStore distributed storage system, which used Byzantine agreement to provide robust protection against data loss. OceanStore required significant computational resources to do signature calculations for the Byzantine agreement algorithms.

Other examples of systems which require significant computational resources are those that are designed for Internet-scale services. These services must scale automatically as load increases, and one class of experiment is the determination of scaling parameters in terms of physical resources vs load. In order to conduct these experiments, a researcher must reserve clusters for the experiment.

A cluster is fairly simple: it is simply a collection of virtual machines and a switching fabric beneath and between the machines. The fabric itself is a virtual LAN, with layer 2 switching supported.

```
<CLUSTER NUM_VIRTUAL_MACHINES = Positive Integer>
     <VIRTUAL_MACHINE>  <! Representative of all VMs →
     <ROUTER>  <! Representative of all Routers →

     <LAN MIN_LATENCY=Unspecified | Real Value in ms
                 MAX_LATENCY=Unspecified | Real Value in ms
                 MIN_BANDWIDTH=Unspecified | Integer Value in Mbs
                 MAX_BANDWIDTH=Unspecified | Integer Value in Mbs
                 LOSS_RATE = Unspecified | Real between 0 and 1.0/>
 </CLUSTER>
```

Note that the LOSS_RATE parameter here is not required for allocation; however, since an experimenter will in general want to specify a loss rate, we permit it to be specified here.

5.5.5 Links

A link is a pointwise connection between two other elements of the system. It simply looks like a LAN with a name, and has guaranteed latency and bandwidth. The endpoints of the link are given by URLs, which are in general interface names from interfaces as specified by virtual machines in the environment.

The only significant distinction between a cluster and a link is that a cluster has its constituents defined hierarchically, and a link has two endpoints, given by name.

```
<LINK MIN_LATENCY=Unspecified | Real Value in ms
                 MAX_LATENCY=Unspecified | Real Value in ms
                 MIN_BANDWIDTH=Unspecified | Integer Value in Mbs
                 MAX_BANDWIDTH=Unspecified | Integer Value in Mbs
                 LOSS_RATE = Unspecified | Real between 0 and 1.0
                 ENDPOINT1 = URL
                 ENDPOINT2 = URL />
```

5.5.6 Routers

A new feature of GENI, not present in Emulab or PlanetLab, is programmability in layer 3. The VINI blade system is an example of the programmable router fabric in GENI.

While this is a conceptually new feature, from the perspective of a resource allocation system a router simply looks like a virtual machine with a different processor architecture, (probably) no disk, and a great many interfaces. The overall routing discipline used, including whether packets or lambdas are routed, is either specified in the link fabric or in the software loaded on the router. In any case, it is not part of the resource description.

```
<ROUTER>
   <AUP = {list of complied AUPs}/>
    <NIC  Number = 0 to N>
          <INTERFACE NAME=URL (OPTIONAL) MIN_SPEED=Integer Value in
Mbs
                                  MAX_SPEED= Integer Value in Mbs>
```

```
                <FIREWALL PORTS={LIST | ALL}
                           PORT_INCLUSION = (EXCLUDE|INCLUDE)
                           RESTRICTION={ACCEPT|SEND}
                           IP_ADDRESSES = {IP_ADDRESS_LIST |
                      EXPERIMENT_ONLY}
                           ADDRESS_INCLUSION = (INCLUDE|EXCLUDE)
                           VALUE=(PERMIT|DENY) />

                 <FIREWALL…/>
             </INTERFACE>
             <INTERFACE MIN_SPEED=Integer Value in Mbs
                                    MAX_SPEED= Integer Value in Mbs/>
               …(to N)
       </NIC>
     </ROUTER>
```

5.6 Development Plan

We next describe how the development of these resource allocation mechanisms can be incrementally staged over time.

- Develop prototype implementations of the Component Manager, Site Manager, Science Board, and Resource Broker components, and command-line clients to exercise these modules. The goal of this initial prototype is to make the functions performed by these modules and the messages passed between them concrete; no specific policies will be implemented (e.g., how the Science Board decides whether to give a user tokens, how a Resource Broker decides whether to give a user a Ticket in return for a token, etc.) The Component Manager will not create slivers on devices in this initial prototype to allow the prototype to run on a single node. However, the prototype will be implemented as a Web Service to allow easy transition to distributed operation.

- Implement several resource allocation policies on top of the prototype framework. A simple initial scheme allows the Science Board to allocate tokens to users arbitrarily, and dictates that the Resource Broker will issue one Ticket for best-effort access to any requested component for each Token presented. A second policy adds resource reservations and batched access: users can ask the Resource Broker for a fixed share (up to 100%) of one or more components now or at some time in the future; if the request does not conflict with an existing request, a Ticket authorizing this access is granted. A third policy varies the number of Tokens required for best-effort or fixed-share access to a component based on the current and projected load on that component.

- Develop appropriate user interfaces (command-line, GUI, and library-based) to allow human and programmatic interaction with the software modules. For example, using these interfaces members of the Science Board will be able to allocate Tokens to users, GMC will be able to set the Resource Broker’s “exchange rate” between Tokens and Tickets, and component operators will be able to configure the Component Manager corresponding to each component they operate.

- Develop appropriate APIs between the Component Manager and underlying devices to allow the Component Manager to automatically discover resources and their capabilities, and to allocate slivers (under both best-effort and reservation schemes) on those devices.

- Integrate the resource allocation framework with an existing resource discovery service, to allow end-to-end discovery and allocation using a single tool. One way to accomplish this integration is to simply modify the resource discovery tool to make the appropriate calls into the resource allocation modules. The resource discovery tool and resource allocation modules will remain physically distinct software components, with communication using standard Web Services protocols, to allow different resource discovery and resource allocation tools to interoperate in the future.

- Develop a decentralized version of the Resource Broker, and demonstrate its applicability to federated resource management scenarios.

6 Experiment Management and Support

For GENI to enable experimental network and distributed systems research, it must be accessible to the broadest set of researchers, including those with a single PI working on a project with a single graduate student in a school with little prior institutional experience in building distributed systems. Such users should find it feasible to map their experiment onto GENI in a straightforward and predictable fashion.

Unfortunately, PlanetLab does not provide much helpful guidance. While PlanetLab has a number of strengths, it is hard to use for the uninitiated. Users often find they need a local “expert guide” who has already tripped over the potholes in getting started. Writing and running a distributed “hello world” program should be a few lines of code, but instead requires, among other things, clicking several hundred times on a web page to request sliver creation on each node, waiting for the system to install sliver access permissions on the node, determining the process for installing custom software on each node, building scripts to monitor node and process failures, building a data distribution system to pipe logs back to the developer’s workstation, etc.

This lack of effective tools places a high bar against PlanetLab’s occasional use, and addressing this in GENI is a high priority. The attendees of both the DC and Chicago open houses advocated making it a requirement that GENI support repeatable, predictable experiments that are easy to set up and configure. Further, we believe this ease of use needs to be available early in GENI’s lifetime, as it will be more difficult to regain the community’s confidence later on. If GENI develops a reputation for being hard to use or configure, the perception will likely stick despite later improvements.

6.1 Types of Experimenters

Before we delve into the requirements of an experiment support toolkit, we first need to examine the kinds of users we would need to support and the types of experiments they are likely to conduct on GENI. We present one such high-level classification below.

Beginners vs. expert programmers: The needs of these two groups of users are distinct. A beginner should be insulated from the myriad operational issues associated with conducting a GENI experiment and is also unlikely to require fine-grained control over issues such as resource allocation. A beginner is also more likely to benefit from the availability of an interactive program, such as a shell, to deploy and control an experiment. An expert programmer, on the other hand, would desire greater control and might want a richer API for controlling experiments.

Short-term experiment vs. long-running services: Short-term experiments are less likely to be affected by exceptional conditions such as resource exhaustion, node failures, and/or dramatic variations in system performance. Long running services, on the other hand, might need mechanisms to identify faults, swap in new resources, and migrate existing programs. A programmer maintaining a long-running service would also be willing to invest effort into building a "control program" to, for example, issue a sequence of management operations in response to failures and other changes in service state. By encapsulating all of this intelligence in a control program, which is constantly monitoring and performing network-wide management, it is possible to deploy long-running services that require minimal manual control.

Homogeneous vs. heterogeneous deployment: Many experimenters might be satisfied with simple deployments, such as allocating any set of ***N*** nodes for the experiment rather than requiring specific forms of connectivity or computational power. Others might have specific connectivity, computational, and environmental requirements, especially since GENI will have different types of resources. The experiment support toolkit should provide a simple API for those requiring homogeneous employments, while also providing the ability to articulate more complex requirements for heterogeneous deployments. In addition to controlling homogeneous and heterogeneous deployments, an experimenter might also request a specific form of emulated network, comprising of an arbitrary network topology and an arbitrary sequence of network events to evaluate new protocols. VINI is such an example platform. The experiment support toolkit should ease the burden of setting up VINI-like experiments.

6.2 Desired Attributes of the Experiment Management Toolkit

Support gradual refinement: We believe that a primary requirement for ease of testbed use is a smooth implementation path from simulation to deployment, with a single, integrated toolset. The developer can start with controlled settings in emulation mode, gradually refine the application to deal with various complexities of the real world, before deployment in the wild, all while using a single set of tools. Emulation mode, provides the benefits of repeatability, performance isolation, controlled faults, and limited dependence on other services, such as resource discovery and allocation. As the application matures, the complexity associated with interfacing with other services could also be gradually introduced, and even performance isolation could be relaxed, e.g., by purposefully introducing jitter or failure.

- **Flexible programming environment**: A beginner might desire a single monolithic tool to address all aspects of experimentation, such as resource discovery, experiment setup, experiment monitoring, and the gathering of results. But with additional experience,, the programmer might desire greater flexibility and not be constrained by tools that prescribe a specific sequence of operations in setting up experiments (such as resource discovery followed by resource allocation followed by service deployment and application control). The flexibility requirement is reflected in a number of design considerations.

- First, each component should be designed such that it could be used stand-alone or composed in an application-specific manner. For instance, a service should be able to invoke resource discovery / allocation / de-allocation at any point during a service's lifetime in response to failures or other events.
- Second, the control toolkit functionality should be available in different forms to support the requirements of a variety of users. The toolkit could be accessible through a library interface (which works with any of the commonly used languages), made available as a shell providing global control of the experiments, or integrated with scripting languages. The success of systems such as Tcl / Tk that integrate domain-specific libraries with a scripting language clearly serves as an inspiration in this regard.
- Finally, where possible, we would like to encourage the use of existing tools instead of inventing new mechanisms. For instance, we would prefer to represent processor attributes in XML and alow the user to employ custom filters (for example, SQL commands) to select nodes. The alternative is to convey user preferences to the runtime system using a specification language, but that approach runs the risk of anticipating all of the selection criteria that a user might employ.

Incremental refinement and integration: The experiment support toolkit could abstract mechanisms and services, e.g., for resource discovery, allocation, and content distribution, as and when they are made available. Ideally, the user can just change an environment variable or a parameter in the API to use different content distribution systems (such as Bullet, BitTorrent, Coral, Codeen) or different resource discovery and allocation mechanisms (such as SWORD, Bellagio, SHARP).

Sophisticated fault handling: On the GENI testbed, it is likely that experiments will face a variety of failures on a frequent basis. Due to resource constraints, experiments frequently exhaust memory, file descriptors, and network-level resources. The experiment support toolkit should enable the gradual “hardening” of distributed services by first allowing service execution in in controlled settings,where faults can be deterministically and gradually administered and programs can be spared from having to face the full brunt of faults. The toolkit should also support common design patterns for dealing with faults. One such component would be a transactional service manager for service deployment. By supporting these abstractions in the toolkit, we hope to ease the development of sophisticated failure-recovery mechanisms.

6.3 Prior Work

Before we discuss what functionality should be present in the experiment support toolkit, we briefly consider the design space of control toolkits for distributed systems.

One option is exemplified by some of the job controllers in the Grid initiatives (such as the Globus controller and GrADS). These systems share many of our goals for configuring and managing the execution of distributed application. These systems provide the "run this experiment" model; they take a user specification, perform resource discovery and allocation, deploy the experiment, monitor its status, and collect the results upon completion -- a fixed sequence of tasks with little room for customizability. Such a scheme is well-suited for running simple experiments and could serve as an appropriate model for designing monolithic job controllers. The various Grid control programs are however not appropriate for long-running network services that might perform iterative resource allocation in response to increased load or failures or where different versions of a network service could be allowed to coexist.

Plush is an experiment management system targeting a variety of distributed computation environments, including PlanetLab and local clusters. Plush provides a framework for performing resource discovery, application deployment, and application management. Plush also provides a shell that could be used to perform system-wide operations. Plush provides a top-down design where a Plush controller performs monitoring and control on an XML application description provided by the user. Plush's specification language serves as a reference design on how to specify the parameters of an experiment. While Plush provides some support for long-running services, for instance, allowing for "re-matching" of resource requests upon failure, the API and callback mechanism would have to be enriched to support the variety of requirements we envision for GENI.

Condor augments the above model with a "checkpoint and restart" facility automatically provided by the runtime system. While Condor provides a form of failure recovery, it is inappropriate for most network services. Consider for instance a program performing network measurements or one participating in a content distribution network. In both cases, checkpointing might not be required. The global controller could either simply ignore the failure or it might perform a restart followed by a call to an appropriate failure handler, as the service might have built-in intelligence for reconstructing its state using application-specific techniques. Long-running network services and other planetary-scale applications require even more flexible error handling and the ability to perform adaptive resource allocation.

A number of tools, including vxargs, Codeploy, and gexec, provide support for remote job execution, but they provide only limited functionality. For instance, these tools do not provide flexibility to control the synchronization between different commands, do not report or react to failures, do not allow the user to control the granularity of application deployment, and they do not provide scalable control for hundreds of nodes.

Similarly, node-side configuration tools, such as cfengine, could be used in conjunction with the experiment control toolkit. cfengine is useful for configuring node state using declarative rules. It does not however provide system-wide control.

The Planetlab Application Manager provides a set of client-side scripts and a server-side web-based interface to deploy an application and monitor its status, especially for long-running services. It however provides limited functionality, as it does not provide means for executing system-wide commands, does not provide a fault-tolerant, scalable control infrastructure, nor does it provide a programming API.

Emulab is a network emulation testbed that runs on a 300-node cluster with network support for traffic shaping (constraints on bandwidth, latency, and loss rate) between arbitrary nodes. Emulab users declare their desired node and network topology using a GUI or *ns* script. A centralized deployment program locates and reserves nodes for exclusive use by the experimenter (Emulab is space-shared), as well as the necessary traffic-shaping nodes. It loads an operating system of the user’s choosing and configures the traffic-shaping nodes and network switches as needed; the user is then responsible for deploying their experiment on the nodes. In “batch mode,” the user submits the code for their experiment along with the topology description. Emulab queues the experiment until sufficient resources are available, at which time it will set up the cluster for the user’s experiment and execute the user-supplied code. Emulab offers some rudimentary support for barriers when starting up applications, but it does not provide facilites for detecting and responding to failures or for collecting experiment logs. While the Emulab software allows users to request, deploy, and monitor experiments, they are designed for running short-term experiments, and are less useful for maintaining long-running services especially in settings where failures and resource fluctuations are common.

6.4 Overview of Recommended Approach

As stated earlier, a key goal for GENI’s experiment support toolkit is to support a smooth implementation path from simulation to deployment, with a single, integrated toolset. In our view, the best way to do this is to target novice users first, and then move to the more advanced support required by power users. Of course, our goal is to design a single integrated system for both, e.g., by building a shell that can be scripted and that can embed extensible API calls. However, we note that novice users are both least well served today (and therefore most open to new toolkits) and have less demanding requirements (lowering the barrier to adoption). Since novice users often start with Emulab, our initial focus is to provide a unified process from Emulab through PlanetLab to VINI and GENI. Just as the UNIX shell enabled unified access to machine resources for both novice and expert users, a similar shell, where individual commands could be applied to different machines within a larger ensemble and with explicit support for partial failure, could dramatically lower the barrier to entry for the novice user while significantly enhancing the abilities of the power user.

We make the following observations on the overall design of experiment support:

- As our primary goal is to smooth the development process by which new ideas go from initial experimentation to eventual deployment, we seek to provide an abstraction layer that allows experimenters to migrate from locally available clusters to planetary-scale testbeds. The API, shell, and integration to scripting languages should work on local clusters, emulation systems such as Emulab / Modelnet, and planetary-scale systems such as GENI. Users can continue to use the same API, shell, or scripts as they migrate. The developer can then start with controlled settings, where there is repeatability, homogeneity, and relative absence of faulty conditions, gradually refine the application to deal with various complexities of the real world, before deployment in the wild.
- The toolkit should export different kinds of interfaces for different types of users. For the new user community, the shell would provide an interactive medium for deploying and controlling experiments. For the existing PlanetLab user community

and for those maintaining long-running services, the API would be made available through scripting languages, so that experienced developers can develop control programs for managing their applications and services. For developers requiring greater control, the API will be exposed in its raw form, allowing fine-grained control over experiment resources.

- The design should allow incremental development of the API modules and incremental access to the newly developed modules through the shell and the scripting languages. The API could thus provide an abstraction layer that allows an experimenter to use different mechanisms or services based on the experimenter's preferences and the availability of services/modules.
- The shell should incorporate some baseline mechanism for resource discovery (with support for later extensibility to other systems) as part of the system architecture. That is, users should be able to issue commands to find subsets of global resources that match some criteria (for instance for network capacity or CPU load). Results of such queries can be used to inform the placement of individual jobs within a larger user application. The shell should also have baseline support for system monitoring, both for failure and for significant changes in available memory, CPU, or network conditions. Once instantiated, the results of the monitoring may be as simple as printing alerts to the shell or to a log file. In a more sophisticated configuration, callbacks may be provided programmatically to an application manager/controller able to reconfigure the application in response to failures or changing system conditions.
- Upon obtaining a desired set of computing resources, the experiment support toolkit must deploy the experiment on the target nodes. The deployment comprises many steps. The target nodes might have to be first customized to execute the application, and this customization task might involve the installation of virtual machines, operating systems, software packages, copying of application binaries onto the nodes, and initialization of background daemons. The system must be prepared for failures among the set of nodes initially made available through the resource discovery and allocation process even at this early stage. Given appropriately configured nodes, the experiment could be started on the nodes and the user would then desire to monitor its execution status. All of these tasks require a powerful set of system-wide job control tools.
- The system should provide simple templates for configuring a distributed application. For instance, there may be common configurations that the system supports "out of the box", such as: i) run a specified application on 50 randomly selected machines and pipe the output back to my local machine, ii) start a long-running service on a specified list of machines, periodically collect the logs (freeing up the remote storage) and provide a callback if one of the application instances or nodes fail, and iii) impose a particular network topology (initially most useful for DETER/Emulab/ModelNet experiments) among a distributed set of nodes and subject the end nodes to a variety of node failures while changing network conditions (bandwidth, latency, loss rate) in a particular pattern. A new user wishing to start

with one of these patterns and perform customization should be able to do so with little overhead.

- The experiment support toolkit should use scalable, robust coordination and communication primitives internally, and should offer those primitives for use by applications. For example, the experimenter should be able to issue asynchronous system-wide commands, query the status of previously-issued asynchronous operations, and synchronize subsequent operations with those initiated earlier. The toolkit should provide various forms of synchronization primitives ranging from traditional barriers to weak partial barriers and point-to-point synchronization.
- One of the primary challenges associated with deploying large-scale distributed systems is the difficulty of debugging a running system. Thus, a requirement for the system is debugging and visualization support for a distributed application as it evolves from a cluster deployment (Emulab/ModelNet) to large-scale wide-area deployment (PlanetLab/GENI). Ideally, it should be possible to single-step a running distributed system and to log sufficient information to step forward and backward for post-mortem, deterministic analysis. In addition, the system should enable sufficient logging to leverage emerging model checkers to interactively explore both actual and alternate system behavior in the face of violations of user-specified system invariants for safety and liveness.

We will now elaborate on the design of the individual components consistent with the above recommendations. We begin by considering the issue of resource discovery, wherein the system attempts to find a set of resources that satisfy an experiment's resource requirements prior to initiating the experiment and then discuss components relating to deploying and monitoring the experiment on the discovered resources.

6.5 Experiment Embedding

Experiment embedding refers to the task of matching the experimenter's abstract description of resource needs to currently available concrete resources. We consider this task for the wired case and will expand this discussion to include wireless and sensor deployments in future revisions of this document.

Our goal is that users should be able to issue commands to find subsets of global resources that match some criteria (for instance for network capacity or CPU load). Results of such queries can be used to inform the placement of individual processes of a distributed application within a large platform such as GENI. At the most basic level, experiment embedding comprises the following three steps.

Monitoring testbed state using sensors and other measurement processes: The monitored state characteristics include CPU loads on testbed nodes, memory usage on nodes, node software configuration, network performance data between pairs of nodes, etc. An important issue is the frequency that various sensors gather the required performance data. To the extent possible, the information should be collected by a single infrastructure and made available to a variety of applications and services that may benefit from it (such as multiple resource discovery services and adaptive applications).

Publishing the data collected by the various sensors: The measurement data could be collected and stored in a centralized node or set of nodes (with replication used to distribute query load and enhance fault-tolerance). An alternative is to store the data in a wide-area distributed data structure such as a DHT, especially if the amount of measurement data cannot be managed by a small number of nodes. However, we expect the amount of measurement data collected to be small enough to be managed by a small number of centralized nodes and that more complex DHT-based solutions are likely unnecessary.

Query processing engine: This component matches experimenters' requests with the available resources and allocates some set of nodes to the experiment. This task is by far the most complex step in experiment embedding. Its design depends on a number of considerations relating to experimenter criteria.

1. Required per-node characteristics are the easiest types of constraints to handle. Examples of such constraints include:
    a. Desired physical location for the nodes (such as North America, Europe, Asia, etc.)
    b. CPU speed, memory size, disk capacity of the desired nodes
    c. Availability of software packages on nodes
    d. Type of Internet connectivity, including access bandwidth
    e. Site-specific constraints, such as the minimum or maximum number of nodes to be allocated from a site
    f. Available CPU, memory, disk, and network resources, given competition from other applications that are using the node

    Note that many of these node attributes are relatively static (items a-e), while some change rapidly (item f).

2. Inter-node constraints are more difficult to handle. Examples include inter-node latency requirements expressed using upper and / or lower bounds, and similar constraints on bandwidth and loss-rates. The most general form of such constraints is an actual topology in the form of desired connectivity requirements. It is well-understood that these types of constraints are difficult to satisfy in the general case, with the topology embedding degenerating into an NP-complete problem.

Given the computationally intractable nature of satisfying fully general inter-node topology constraints, we seek to take advantage of the requirements of typical GENI experiments to simplify the problem. We conjecture that there is likely to be a correlation between the size of an experiment and the amount of control over resource selection required by the experimenter, particularly with respect to control over inter-node constraints.

For example, in a small-scale deployment, the experimenter might desire a high degree of control. Such control is particularly useful for experiments that study new network protocols where the experimenter might want to study the dynamics of the protocol over fine-grained variations in topology and network connectivity. Such experiments are valuable as they would serve as a more convincing form of validation after *ns* simulations and Emulab or Modelnet experiments.

On the other hand, for a large-scale deployment such as a long-running publicly-accessible service, the required degree of control over topology and inter-node connectivity is likely to be less for a number of reasons. First, the experimenter is likely deploying a real service architected to exploit all available resources, in contrast to earlier stages of application development such as sensitivity analysis that requires fine-grained control. Second, specifying a specific topology for a large application is an onerous task. Third, even if the experimenter is willing to undertake the task, it is not clear what constitutes a "good" or "realistic" set of constraints to impose on the embedding.

We therefore conjecture that typical large-scale experiments issue fairly simple queries such as:

- All nodes that satisfy some per-node constraints
- All nodes that satisfy the per-node constraints and no two nodes at the same site
- Some ***k*** nodes that satisfy the per-node constraints, etc.

There are of course exceptions to the above generalization. For instance, if the experiment is to try to deploy a "virtual ISP" and if the experimenter wants to pick ***k*** nodes such that they are well-distributed on the Internet, then the user might desire a great degree of control on the target topology. But we expect such uses to be the rare ones. Our recommendation is that the experiment support toolkit should not directly support uses where optimal large-scale embeddings are desired, but instead should make all resource characteristics readily available to applications. The applications can then use this information to choose the resources using application-specific heuristics when fine-grained control is required for the largest deployments. Solving the topology embedding problem for smaller deployments should be computationally feasible using a variety of available approximations to the general NP-complete problem.

Given the above discussion, we make the following recommendations for implementing the experiment embedding query processing engine:

1. Support all forms of per-node constraints.

2. Support simple group constructs, such as allocate on all nodes, one node at each site, or some subset of nodes / sites.

3. Provide a brute force method for embedding requests with inter-node constraints for small-scale experiments. Clearly, the brute force method (using exhaustive search) is feasible only for small-scale embeddings. The brute force method should be made available both as an online service and as a library that can be linked with an application deployer's resource selection code and run locally. Since the brute force method could be computationally expensive, the service needs to execute embedding requests with limits on computational load per query. For those applications that might require embeddings for more than a small number of resources, the experimenter should use the library version, which means that the computational load could be offloaded to the experimenter's local machine.

4. In addition, make all of the sensor data and node configuration data available to the application so that the application can decide which resources it wants to use instead of requiring the experiment embedding system to identify the appropriate set of nodes.

We conclude our embedding discussion with a few issues that concerns its integration with the rest of the system. Clearly, a resource allocation broker needs to be integrated with resource discovery because the embedder might provide a solution containing some resources that the resource broker cannot obtain. In such cases, the resource broker would have to invoke the embedder again to obtain a different solution to the resource discovery problem with the added constraint that some set of nodes should not be considered in the subsequent query. This interaction is particularly critical in a reservation-based resource allocation system if the resource allocator is unable to negotiate the reservations. It is conceivable that resource discovery and allocation could be perhaps more easily decoupled in a best effort environment where access to all resources is always permitted.

Another dependency occurs during program execution. The program would typically have to be executed with a certain set of certificates regarding its privileges -- such as whether the program can access the network or not. Instead of discovering these certificates on-demand, they could be part of the information a node advertises to the resource discovery service. If such a binding exists, then we could have the process execute with the appropriate set of certificates readily and locally available.

6.6 Enabling Robust Execution

- To support permanent services and other advanced applications, the toolkit should include mechanisms for continuous monitoring of program state and reporting exceptions and faults to the user. The same infrastructure used to support resource discovery can also support more general system monitoring, both for failure and for significant changes in available memory, CPU, or network conditions. The results of the monitoring may be as simple printing alerts to the shell or to a log file. In a more sophisticated configuration, callbacks may be provided programmatically to an application manager able to reconfigure the application in response to failures or changing system conditions.

- The task of building such adaptive applications can be made easier with support from the experiment support toolkit for common design patterns that application developers use for recognizing and overcoming faults. It would be useful to provide transactional operations or operations with “executed exactly once” semantics. If transactions fail, the target nodes could be (temporarily) excluded from the experiment. We will allow users to specify what to monitor to recognize failures (such as running processes) and what actions to take when faults occur. When a system-wide command times out on a node, the failure could be reported to the controlling entity. In the absence of a centralized monitor, the failure may be reported to some subset, or possibly all, of the nodes running the application and what coordinated corrective actions need to be taken, such as executing a leader election protocol.

- In addition to monitoring the status of the executing program and reacting to faults, it is also desirable to have intrusive and non-intrusive techniques for monitoring program state, detecting abnormal behavior, and debugging support (such as single-stepping). Debugging mechanisms could be used to register commands in a node repository that node programs can access using library calls. These commands could be used to issue requests to stop the process, perform debugging or checkpoint application state. Node programs could also register handlers to be invoked for handling failures and exceptions. These handlers could then be invoked asynchronously when the monitoring program detects these anomalies.

6.7 Scalable and Fault-Tolerant Control

Comment [AV1]: It seems like we could beef this subsection up with additional discussion. This is a fairly critical/tricky requirement.

- The control infrastructure needs to scale to hundreds and thousands of nodes without developing hotspots. Performance hotspots are to be particularly avoided while sending control commands or distributing content. Control signals, monitoring, and data gathering therefore needs to be performed over an overlay tree or a mesh to achieve scalability. A mesh-based overlay could be used for rapidly and redundantly transmitting control information among nodes participating in the experiment. Note that such an infrastructure could be shared among all experiments/services running on GENI instead of having a per-experiment instance. For example, the same mesh could be used by non-management applications such as content distribution applications. Reliability can be enhanced by using a resilient communication layer that routes control messages around network faults and hides transient connectivity problems (as in MIT-RON).

6.8 Development Plan

We next describe how the development of the experiment support toolkit might be broken down into quantized components and staged over time.

- Develop and implement an API for experiment instantiation and system-wide job control using some parallel execution mechansim. Develop a shell program to allow interactive invocation of the tasks provided by the API. Demonstrate that a new user can quickly and easily deploy experiments on Emulab, PlanetLab and GENI using the shell.

- Develop support for using the API with scripting languages, such as Python and Perl. Develop more advanced support for parallel process management, including suspend/resume/kill, and automated I/O management. Also develop support for issuing system-wide commands in asynchronous mode, querying the status of previously issued asynchronous operations, and to synchronize subsequent operations with those initiated earlier. Provide support for simple forms of node selection, e.g., based on processor load or memory availability. The goal is to provide the abstraction of "run this experiment on some set of nodes."

- Develop support for global synchronization: Following along the lines of Plush, the toolkit should provide various forms of synchronization primitives for use during program execution. Synchronization primitives should span the entire range from traditional barriers to weak partial barriers and point-to-point synchronization.

- Develop support for permanent services. Develop mechanisms for continuous monitoring of program state and reporting faults to the user. Develop support for deploying experiments on heterogeneous GENI resources. This would include support for specifying network configuration, controlling (or injecting) network events, exporting information regarding network conditions, and providing more control over resource allocation for a diversity of resources. The goal is to provide the abstractions of "keep this service running" for service developers and "run this experiment on a matching set of nodes/topologies" to network experimenters desiring a specific system configuration.

- Develop algorithms for resource selection: For users desiring specific topologies, potentially specified by end-to-end performance requirements among every pair of nodes, simple resource discovery schemes might not suffice. Instead, one requires algorithms, and heuristics to locate topologies that match user requirements. These algorithms could be based on the resource allocation algorithm used by Emulab's **assign** utility or the test-bed embedding algorithms proposed by Considine, Byers, and Mayer-Patel.

- Interface the experiment support toolkit with other services described in this document, such as CPU performance monitoring and network performance monitoring. Also provide interfaces to different resource discovery and allocation mechanisms (such as SWORD, Bellagio, SHARP), and different content distribution systems (such as Bullet, BitTorrent, Coral, Codeen), so that the user can just change an environment variable or a parameter in the API to use these services.

- Develop support for fault-tolerant execution: In particular, develop support for commonly used design patterns such as transactional operations, two-phase commits, or operations with "executed exactly once" semantics.

- Develop intrusive and non-intrusive techniques for monitoring program state, detecting abnormal behavior, and debugging support such as single-stepping.

- Address scalability issues so that the control infrastructure can scale to hundreds and thousands of nodes without developing hotspots. Control signals, monitoring, and data gathering needs to be performed over an overlay tree or a mesh to achieve scalability.

- Address network reliability issues by having the monitoring and control infrastructure use a resilient communication layer that routes control messages around network faults and hides transient connectivity problems (as in MIT-RON).

We also plan to consider including support in GENI for additional tools such as language and compiler support, program and protocol analysis tools, and methods for program verification and validation.

7 Operations Support

Monitoring and operations support have two main goals: the first is to reduce the cost of keeping the system running. The mechanism by which we expect to impact the operations cost is automating many of the tasks required of the operations staff. The second goal is to provide a mechanism for gathering data on the operation of the system. As part of this goal, we also expect that the monitoring system will provide a convenient way to query or disseminate this data. This support is useful to verify that the system is operating within expected tolerances, and for researchers to observe how the load on the system may be impacting their own services, experiments, or measurements.

In this section, we provide an overall philosophy of operations support and monitoring, with some concrete descriptions of the features such a system should provide. We do not attempt to itemize every prescriptive check possible, nor do we undertake a complete survey of the requirements of the operations staff for managing GENI. Our observations are largely based on historical information from the operation of PlanetLab. We focus on two aspects of the problem: providing a general-purpose monitoring framework, and using monitoring to reduce operational support costs. Some of the systems we discuss below have been published, while others are also operational and available to the public Internet. Additional information chronicling real operational problems experienced in PlanetLab can be found in [ADA03].

7.1 Monitoring

Monitoring in GENI serves several purposes: detecting activity and problems on the nodes, determining which projects are responsible for any peculiar behavior on the nodes, and providing some feedback regarding resource usage. It should be noted that monitoring and auditing, while related, serve separate functions. While auditing is a critical function for reporting problems, monitoring is often just a desirable service. However, an easily-available and widespread monitoring service also makes it easier for individual projects to identify problematic behaviors and make themselves more robust. Several monitoring projects have existed on PlanetLab, including Ganglia [MAS04], Trumpet [BRE04], and CoMon [PAR06]. We intend to combine aspects of these systems that make monitoring easier.

Ganglia is widely used in monitoring clusters, and allows aggregation of data at various scales, enabling data reduction while displaying metrics. This aspect of the service is important for providing summary information about a large number of systems in a small amount of screen real estate. Trumpet provides a number of end-to-end tests to provide a form of continual QA of its environment, and it provides a repository that allows other projects to gather this data for use in node selection, node avoidance, and slice embedding decisions. Trumpet is also the only monitoring project (of the ones discussed here) that uses a decentralized infrastructure. While a completely decentralized infrastructure has caused it some scalability problems, it has also resulted in the data being more available than any site on which it runs. Finally, CoMon uses a table-based representation to provide information about the state of PlanetLab, allowing a large amount of information to be conveyed, but also requiring some significant screen real estate. Like Ganglia, it can provide graphs of its data, but CoMon's graphs tend to be simpler and single-valued, whereas Ganglia often combines multiple related values per graph.

These systems collectively have provided some experience on what features are useful in monitoring systems

- Provide an easy-to-use Web interface with public access, because much of the initial use of these systems tends to be performed by researchers trying to debug performance anomalies in distributed applications.
- Support some sort of query language that allows data extraction from the monitoring system. This approach allows the query system to be used when controlling applications, or when launching new applications from scripts. The query language can range from simple to expressive. The CoMon query language consists of C-like mathematical expressions to select nodes that match the constraints. The query language for SWORD uses an XML syntax and allows more open-ended optimization constraints.
- Maintain archives of the data, and make them easily available. Much of the interest in these systems comes from understanding the behavior of applications over longer time scales, or the behavior of the testbed as a whole.
- Flexibility in the data output – a rich output format that can be extended over time has been extremely useful in CoMon, and also appears in Ganglia. The difference between the two systems is that CoMon uses an HTTP-like format for all data, while Ganglia uses a binary format for its own data and XML for administrator-extended data. From lessons with CoMon, text formats seem preferable, since they can be easily used in a variety of languages and tools. Whether the format chosen is XML or something else probably does not make much of a difference.

Given that a number of the above-mentioned systems have non-overlapping measurements, it is desirable to provide a base architecture that can flexibly accommodate a variety of diverse data sources and support a wide variety of user needs. We propose a model using techniques commonly found in service-oriented architectures. For each element to be monitored, the monitoring system should allow a number of different data sources to be queried using HTTP, and the sum total of these responses will be available as the raw data about the state of the element. For example, data for a node may be available as http://fully.qualified.node.name:port/sensorname, where the port and sensor name provide a specific measurement gathered on the element. Note that in the case of low-capability monitoring elements, some of these requests will actually be handled by the gateway to the element itself. Additionally, off-node measurements can also be incorporated, such as http://repository.example.com/query.cgi?nodename, which provides a mechanism for third-party data sources to be incorporated into the monitoring framework. Since the main repository of URLs for querying the data will be administered by the GENI operations staff, we do not need to address the question of malicious URLs being added. Each URL will provide one or more data items in its own name space, with a qualifier associated with the providing URL. For simplicity and to leverage the existing breadth of tools, all of these data sources should provide data in XML format.

The data will be available in a number of formats

- A tabular HTML format, similar to the format used by CoMon, in which rows of a table represent data elements, and where columns are the values of specific measurements on that element. The table itself will be customizable, such that the client can provide an XML file that describes the layout, position, and contents of the columns, in order to get a table that provides only the data of interest to the client. For general-purpose monitoring, and as a starting point for customized solutions, the monitoring system will provide a base XML file that provides one or more commonly-used views of the data. For example, one monitoring view might include data on all of the hardware in the system, such as disks, memory, CPUs, and network cards, and their utilizations. Another view might identify which slices are the most heavy consumers of each of these. Yet another view might examine base operating systems characteristics, combined with some synthetic measurements of performance.

- This data will also be available as raw XML so that applications that want the data without either screen-scraping or querying the sources directly can simply get it from the monitoring system. By providing the XML schemas (e.g., XSD) as well as the DTDs, the system will be usable by tools like XQuery, etc. The central repository and monitoring service itself should likely support (a possibly rate-limited) XQuery facility hosted on its own hardware, to reduce the need to transfer large quantities of raw data only to have it be distilled dramatically at the client side.

- The monitoring facility should support the ability to automatically detect the violation of any number of configured invariants. We expect that these invariants will come from both the GENI operations staff, as well as the researchers using GENI. The researchers may have invariants such as checking that the number of processes in their slice is less than an expected ceiling, or that the physical memory consumption of their slice is less than a socially-accepted limit. The operations staff may care less about these resource consumption details, but may care about how they interact with the behavior of the node. For example, the operations staff may want to ensure that each slice is receiving service within a certain timeframe, or that the amount of available CPU on each node is above some accepted floor. In case these invariants are not met, the system should be able to take some specified actions. For example, researchers may be periodically e-mailed about their slices exceeding their expected resource consumption, or the slice may be reset if the resource consumption rises beyond a hard limit. In the case of the operations staff, some violations of the invariants may trigger escalating actions – at a predefined level, it may simply become an alert message for the operations staff, while at a higher level, it could cause the shutdown of a slice or the reboot of the affected node, plus a message to the responsible authority at the researcher's site.

7.2 Operational Issues

Beyond using the monitoring systems for simply observing the deployment, we expect that the operations staff can heavily use the invariant-violation detection systems and build upon those to produce higher-level fault analysis and recovery. This approach may require the use of large numbers of measurements that are otherwise uninteresting from a resource monitoring standpoint. For example, the system may periodically check whether the ssh daemon is seeing activity, or if the measurement port has been queried recently, or even if the disk can be written, and how long it takes to perform a disk write. These bits of data will generally be interesting as thresholds – we do not care if the most recent ssh activity occurs in the past minute or 10 minutes ago, but we do care if we see no new ssh sessions in the past hour, if we commonly see them every 10 minutes. Likewise, if the monitoring port has not been polled recently, it could indicate that a centralized monitoring service is down, or it could indicate that some intermediate box is filtering the requests (which has been seen on PlanetLab). We expect that the operations staff will write a large number of these unit tests, and that they run frequently so that regressions can be spotted.

The mechanism for spotting these anomalies will be the use of invariant detectors, coupled with a custom reporting page. Additionally, several monitoring vantage points and protocols can be used so that disconnections in the network, or filtering of specific protocols, does not impact data collection. For example, even if the raw sensor is reporting via HTTP, the monitoring system can also gather this data by performing an ssh to the node and accessing the sensor via the loopback interface.

The operations staff can also automate the monitoring system such that known problems can be automatically diagnosed and corrective actions taken using administrator-supplied scripts. For example, if port filtering is detected by comparing locally-available ports with those remotely accessible, a first-level mechanism to address the issue may be to send e-mail to the local site administrators. After a period of time, the issue could automatically be escalated to the site PI, and then if no effort is still taken, slices affiliated with the affected site could revoked at some point. Likewise, this system could be used to handle machine failures that require hands-on assistance – if no progress is made on automated problem tickets, the site could be blocked from using resources at other sites. As experience with GENI progresses, and as the monitoring data spurs more research, more complicated root-cause analysis schemes can be envisioned. For example, if several slices are behaving erratically on one node, and are normal on other nodes, we might immediately suspect that the node is at fault, rather than the slices. The converse is also true – if a set of nodes is behaving erratically, and that set overlaps with the deployment of a particular slice, we may suspect that the slice's behavior is affecting the nodes. It may be possible to correlate the onset of anomalous behavior in the operating system with the behavior of slices, such that evidence from multiple nodes could be used to reduce the uncertainty of any particular observation.

Longer term, this system could also be used in conjunction with the resource management system – if a slice normally follows a particular resource usage profile, and its resource consumption suddenly deviates from its known behavior, it may be a sign that the slice has been compromised. In these cases, the monitoring system could alert the researchers involved, and even disable the slice automatically if no response is received within a given timeframe.

These examples are intended to illustrate some of the mechanisms that the monitoring system can provide to reduce the support overhead, and to automate many of the tasks that the operations staff would normally provide manually. We expect that as long as the base mechanisms are provided, that over time, the higher-level systems can be built using the observations of the running system.

8 Supporting Legacy Applications in GENI

8.1 Goal

The goal of this section is to foster the use of GENI by end users. Since end users today rely on a suite of applications, services and information sources provided on top of the Internet, we believe a key step is to make it easy for these legacy applications and services to take advantage of new functionality provided by new architectures on GENI. The term "legacy applications" refers to existing applications such as web browsers, databases, chat clients, remote file access applications, etc. Similarly, the term "legacy service" refers to existing services that are invoked by legacy applications through a standard interface. Examples of such services include DNS, email, Internet's data delivery (invoked either via IP or the socket interface).

Our main goal is to develop "compatibility frameworks" which allow the researchers to easily extend the functionality of existing services or deploy new functionality altogether without worrying of supporting legacy applications. Such frameworks would significantly lower the development cost and ensure meaningful comparisons between different implementations of the same functionality / service. Furthermore, these frameworks would lower the barrier of acceptance of new functionalities by real users, and allow the researchers to evaluate their designs in the presence of real applications and real users.

Today's examples of such frameworks are Click, which allows developers to easily modify and add data-path functionality (such as better buffer management, packet scheduling, and packet inspection), XORP, which allows developers to experiment with new routing protocols, and OASIS / OCALA which allow the developer to experiment with new network architectures and overlays.

8.2 Requirements

Here we present a short list of requirements that such compatibility frameworks should meet. Again, the main goal of these frameworks is to ensure that researchers can easily improve, modify or add functionality to existing services while supporting legacy applications.

1. Transparency: Ensure that legacy applications are oblivious to a new architectures or an implementation of a given service. Ideally, legacy applications should work without any changes or re-configurations. Furthermore, the design of the compatibility frameworks should avoid as much as possible changes to the OSes and work with the major operating systems (e.g., Linux, OS X, Windows).

2. Expose new functionality: While in some cases a new architecture or implementation just improves the existing services (e.g., using DHTs to improve DNS scalability), in other cases a new implementation or architecture adds new functionalities to an existing service. For example, Click can be used to add QoS to IP data forwarding, RON to perform routing optimizations for a given performance metric, and i3 to support mobile hosts. The users should have the ability to configure these services to take advantage of their new functionality. Usually, such configuration takes place out-band as the applications themsleves are oblivious to the new functionality (see requirement (1)).

For example, OASIS and OCALA enable users to take advantage of the functionality provided by new network architectures through special configuration files, where the application taking advantage of a new functionality is identifioed either by an (IP-address, port-number) pair or DNS names.

3. Concurency: A framework should enable multiple instantiations of the same service to run simultaneously. For example a framework designed for the IP service should allow a web browser to use IP to connect to a CNN server, a chat client to use i3 to preserve its anonymity, ssh to use RON for improved resilience, and a web server to use a capability-based architecture to defend against potential DDoS attacks.

   Furthermore, a user should be able to chose what service instance to use on a per application basis. For example, a user should be able to select CoDNS to resolve the DNS names for the GENI hosts (e.g., name of the form *.geni.org) and the existing DNS infrastructure for all other names.

   The ability to run multiple instances of a service concurrently and select any of these services on an application basis has two benefits. First, it lowers the barrier of users experimenting with new service instances, as they have the flexibility to choose when and for what applications to use a specific service instance. Second, it makes it easier for researchers to evaluate different instantiations of a service, as they can run these instances side-by-side.

4. Composability: Allow applications to compose the functionality provided by different service instances. In the case of the IP service, a user may stitch together two different overlays to take advantage of their functionality. For example, a user may use i3 to access NATed hosts and provide end-host mobility at the edge of the network, and use RON to provide superior resilience in the wide are network.

   Note that in the case of the IP service, the composability may allow users to access (and being accessed from) hosts that do not participate in the overlays. This is possible because the Internet is just another instance providing the IP service. This way a user could access the CNN site despite the fact that CNN does not participate in the overlay. Similarly, a web server that uses an overlay for better security can be accessible by clients that do not participate in the overlay.

5. Robustness and security: The compatibility framework should have minimal impact on the application robustness and should not compromise the user security. If a particular service instance crashes, this should not impact applications using other service instances running on the same machine. Also, an application should be able to default to another service instance, when the one is using has crashed.

   The users should not be exposed to the security vulnerabilities potentially introduced by a new service instance. For example, an initial implementations of i3 allowed attackers to easily eavesdrop legitimate traffic. In this case, the design of the compatibility framework should provide the ability to authenticate end-hosts and encrypt the traffic, while supporting the additional functionality provided by a particular architecture. For example, in the case of DOA and i3, the security model should provide support for middleboxes.

9 Related Work

This section is intended to be a placeholder for a thorough discussion of existing standards and software modules that we can leverage to build GENI. Because software is costly to construct, our default position is to leverage off the shelf components wherever possible, and only build where it is essential to meeting GENI's core mission.

For now, this section focuses on web services and related technologies; the reader is invited to suggest additional technologies we should consider for evaluation. Please note that this section may be moved into a separate document.

9.1 Web Services: Introduction

"Web Services" is a loose term applied to the large set of protocols and standards that have arisen to structure interactions with programs ("services") and content available via the HTTP protocol. They range from formatting XML documents according to a set standard to convenient support for cross-platform RPC using HTTP and XML.

Web Services have become extremely popular, and form the basis of a number of real, complex distributed systems. They are in common use inside Amazon.com, for instance, to link together services such as catalog lookups, advertising, reviews and recommendations.

Despite their recent popularity, Web Services offer little that is fundamentally, technically different from existing RPC systems or techniques for remote data access. Where they differ is that the typical Web Service operates at a much higher level of abstraction than, for instance, a typical SunRPC service. The level of function offered by earlier RPC systems more closely matched function call boundaries; Web Services, in contrast, more often encapsulate entire services (e.g., create and display a map of a particular region, or return the results for a search term). The distinction is not large, but in practice, it meshes well with a trend towards the use of higher-level languages (ruby, python, perl, etc.) for rapidly creating complex networked services.

The rest of this section provides an outline of the available Web services and then discusses specific recommendations for their use (and dis-use) within GENI.

9.2 Available Web Services

9.2.1 RPC-like Protocols

REST is the simplest of the RPC-like protocols. It stands for ``Representational State Transfer.'' It's a fancy name for accepting well-defined parameters in an HTTP GET request and returning a consistently formatted XML document. Unlike more complex RPC mechanisms, REST specifies no data types or representations, and there is no standard for how to specify a REST page. An example REST service is Yahoo's web services APIs, which specify a request URL, the names of the parameters that can be passed, and an XML Schema for the names of the response elements.

REST is an improvement upon poorly structured, HTML-formatted web forms that require ``screen scraping" (generating a specific parser to extract results) to access from a program.

REST fits in well with the w3c's RDF (``Resource Description Framework"), an XML-based specification for describing resources and information. A common example of RDF is its use inside a card catalog that describes books or web pages. Like REST, RDF stresses making web data available in a well-structured manner so that it can be interpreted by programs, not just by people. Unlike REST, RDF has less of an implication that it is being used in an RPC-like manner. An example use of RDF within GENI would be its use in specifying a list of GENI nodes and their attributes.

SOAP is a heavyweight RPC protocol that operates over not just HTTP, but also SMTP, Jabber, and other protocols. While it is most commonly used as a synchronous RPC mechanism over HTTP, it can be used as a more general, asynchronous message-based system. SOAP provides an extensible namespace for introspection, parameter and result definition, and creating datatypes.

A drawback to SOAP is its verbosity. For example, a one-parameter RPC consumes about 408 bytes:

```
SOAP-ENV:Envelope

 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"

 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"

 xmlns:xsd="http://www.w3.org/1999/XMLSchema">


  <SOAP-ENV:Body>

   <ns1:getTemp

    xmlns:ns1="urn:xmethods-Temperature"

    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

     <parameter xsi:type="xsd:string">param-value</parameter>

  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

XMLRPC is the final choice for Web Services RPC. It is somewhat less extensible than SOAP, is designed exclusively to operate as an RPC protocol over HTTP, and is much less verbose (though it is still bulky compared to traditional, binary RPC mechanisms). It lacks many of SOAP's extensibility, such as namespaces. A simple single-parameter XMLRPC call is 159 bytes:

```
<?xml version="1.0"?>

<methodCall>

  <methodName>examples.getStateName</methodName>

  <params>

    <param><value><i4>41</i4></value></param>

  </params>

</methodCall>
```

Because of its verbosity and flexibility, parsing SOAP is a moderately heavy-weight operation. XMLRPC is easier to parse (e.g., you can do it in Javascript). REST can be either, but its back of a clear schema makes it difficult to write automatic stub generators as can be done with other RPC mechanisms. There is good library support for both XMLRPC and SOAP. Much of the distinction between then depends on the application.

9.2.2 Service Description: WSDL

WSDL is an XML-based service description language. It's much like a traditional IDL, such as the RPC Language used by, but it's an IDL that no human wants to type in. For example, the WSDL for a simple one-parameter service is nearly 800 bytes long. WSDL is very flexible in what it can define, though most current uses are for fairly basic RPC. WSDL is supported by SOAP.

9.2.3 Naming and Location: UDDI

UDDI, or ``Universal Description, Discovery, and Integration'', is a directory for WSDL-described services. It runs on top of SOAP to provide bindings between a more abstract service description (e.g., ``something that implements a particular interface'') and a service that provides that interface. For example, a common UDDI theme is to:

- Publish a UDDI standard for a service (e.g., flight reservations)
- Have orbitz, etc., register into a UDDI directory

- Clients can walk / look up in directory to find desired services, automatically.

The UDDI description is unfortunately replete with marketing. The founders view was that UDDI would be the basis for a universal business directory, but the real world use so far is limited to a service directory for indirect binding of services to implementations and instantiations. UDDI doesn't appear to have the momentum that other web services (SOAP, XMLRPC, WSDL) do.

9.3 Web Services in the Real World

- Google search API: SOAP
- Google maps API: Custom javascript + REST
- Yahoo APIs: (many): REST
- Emulab, Planetlab. OÕreilly: XMLRPC
- Flikr: SOAP, XMLRPC, REST

In other words, there is still no clear single standard for web services, and each technique has its advantages and disadvantages, promoters and detractors. Fortunately, it is not difficult to support multiple access methods should it become necessary or useful to do so. The real difficulty lies in creating the service itself; granting remote access to it is usually a simple step thereafter.

9.4 Event Notification: RSS and Atom

RSS and Atom are primarily *formats* for making event notification data available, not mechanisms for efficiently distributing pub/sub data. This is still an emerging area, and it's likely that the real value arising here will have less to do with the particular format and more to do with the services that emerge to distribute them.

9.5 Summary

Web services are extremely easy to use with high level languages, and can make it possible to easily construct and compose powerful services. Examples abound, for instance, of merging Google's maps with data from other sources (such as apartments for rent on the popular Craigslist service). They have enormous momentum, and are available on most platforms and with bindings for most popular languages.

The drawbacks to web services are primarily ones of performance. They impose high processing overhead (XML processing, the SOAP name spaces and attribute parsing, etc.). As a result, they have relatively high latency, large message size, and are moderately complex. Certain advanced aspects of web services, such as multicast and specifications for failure handling, reliability, and response time, are still poorly defined.

As a result, the best places to use web services within GENI are for handling high level operations, and those that present ``human"-level data:

- Return a list of 300 GENI nodes matching a specific criteria
- Draw a map with this set of nodes overlaid
- Return the node use quota for user U

The ease of use of Web Services makes such high level services extremely attractive for further composition and reuse, and their relatively coarse-grain means that the overhead imposed by the web services protocols will not be prohibitive. Web services and RDF should be *strongly* favored over simply providing poorly-structured HTML, which cannot be easily interpreted by programs.

Areas where web services are less likely to work well include systems software services in which performance is a concern, and those with non-standard interfaces, such as a distributed hash table (an application in which the service developers are likely to have to construct a lot of mechanism for routing, etc., already). In these respects, the venerable XDR / sunRPC is still extremely functional, and web services offer few benefits.

Finally, event notification services such as RSS and Atom are still emerging; while solutions may emerge here, this is still an area of active research and should be watched before GENI adopts them for uses outside what is in common parlance today.

References

[ADA03] R. Adams. PDN-03-015, Distributed System Management: PlanetLab Incidents and Management Tools, November 2003

[AUY04] Alvin AuYoung, Brent N. Chun, Alex C. Snoeren, and Amin Vahdat, Resource Allocation in Federated Distributed Computing Infrastructures, Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure, Boston, MA, October 2004.

[BRE04] Paul Brett, Rob Knauerhase, Mic Bowman, Robert Adams, Aroon Nataraj, Jeff Sedayao, and Michael Spindel. A Shared Global Event Propagation System to Enable Next Generation Distributed Services. Workshop on Real, Large Distributed Systems. San Francisco, 2004.

[CHU03] Brent N. Chun, Yun Fu, and Amin Vahdat, Bootstrapping a Distributed Computational Economy with Peer-to-Peer Bartering, Proceedings of the First Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, June 2003.

[CHU05] Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, and Amin Vahdat, Mirage: A Microeconomic Resource Allocation System for Sensornet Testbeds, Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II), Sydney, Australia, May 2005.

[FU03] Yun Fu, Jeffrey S. Chase, Brent Chun, Stephen Schwab, and Amin Vahdat, SHARP: An Architecture for Secure Resource Peering, Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP), Bolton Landing, NY, October 2003.

[GDD-06-07] GENI Planning Group, “GENI: Conceptual Design, Project Execution Plan,” GENI Design Document 06-07, January 2006.

[GDD-06-08] Larry Peterson (Ed), “GENI Design Principles,” GENI Design Document 06-08, GENI Planning Group, March 2006 (updated August 2006).

[GDD-06-11] Larry Peterson (Ed), “Overview of the GENI Architecture,” GENI Design Document 06-11, 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.

[MAS04] Matthew L. Massie ,Brent N. Chun , and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Parallel Computing, Vol. 30, Issue 7, July 2004.

[NG05] Chaki Ng, Philip Buonadonna, Brent N. Chun, Alex C. Snoeren, and Amin Vahdat, Addressing Strategic Behavior in a Deployed Microeconomic Resource Allocator, Proceedings of the Third Workshop on Economics of Peer-to-Peer Systems, Philadelphia, PA, August 2005, pages 99-104.

[PAR06] K. Park and V. S. Pai. CoMon: A Mostly-Scalable Monitoring System for Planetlab. *Operating Systems Review* , Vol 40, No 1, Jan 2006.

[RAG04] Barath Raghavan and Alex C. Snoeren, A System for Authenticated Policy-Compliant Routing. Proceedings of the ACM SIGCOMM Conference Portland, OR, September 2004, pages 167-178.

[SCH05] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent N. Chun. Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems, Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X), Santa Fe, NM, June 2005, pages 37-42.