Designing a Federated Testbed as a Distributed System

Robert Ricci[1], Jonathon Duerig[1], Leigh Stoller[1], Gary Wong[1],
Srikanth Chikkulapelly[1], and Woojin Seok[2]

[1] University of Utah, School of Computing
ricci,duerig,stoller,gtw,srikanth@cs.utah.edu
[2] Korea Institute of Science and Technology Information
wjseok@kisti.re.kr

Abstract. Traditionally, testbeds for networking and systems research have been stand-alone facilities: each is owned and operated by a single administrative entity, and is intended to be used independently of other testbeds. However, this *isolated facility* model is at odds with researchers' ever-increasing needs for experiments at larger scale and with a broader diversity of network technologies. The research community will be much better served by a *federated* model. In this model, each federated testbed maintains its own autonomy and unique strengths, but all federates work together to make their resources available under a common framework.

Our challenge, then, is to design a federated testbed framework that balances competing needs: We must establish trust, but at the same time maintain the autonomy of each federated facility. While providing a unified interface to a broad set of resources, we need to expose the diversity that makes them valuable. Finally, our federation should work smoothly in a coordinated fashion, but avoid central points of failure and inter-facility dependencies. We argue that treating testbed design as a federated distributed systems problem is an effective approach to achieving this balance. The technique is illustrated through the example of *ProtoGENI*, a system we have designed, built, and operated according to the federated model.

1 Introduction

Testbeds for networking and systems research have traditionally been built as stand-alone facilities where each testbed is operated and managed by a single entity. The problems of building, running, and improving each individual testbed have received attention in the literature in preference to issues of coordination, trust, and cooperation between testbeds [30, 18, 17, 1].

Increasingly, experimenters need to run larger experiments incorporating a broader diversity of devices and network technologies. It is difficult to satisfy this requirement with isolated testbeds, since each testbed is limited in size and tends to concentrate on a particular type of resource. If experimenters were able to treat a collection of testbeds as a single facility, this would enable them to run larger experiments and take advantage of diverse resources.

This leads to a *federated* model, in which individual testbeds work together to provide their users with a common framework for discovering, reserving, and using

resources. This common framework must meet a number of requirements. It must establish trust between federates, but allow each member of the federation to retain autonomy; each federate should have independent local control over usage policies and resource maintenance. The federation should support pre-existing testbeds, which are managed using a variety of software suites, and which were created to manage different kinds of resources. While accommodating this heterogeneity, the federation must present a single interface and provide the appearance of a single "virtual facility," giving users access to a richer set of resources than any one facility can provide by itself. Finally, the federation must provide coordination among members without sacrificing robustness, as a complex distributed system introduces many points where failures can occur.

This paper makes three contributions. First, it defines a set of five design principles that, together, meet the requirements of a federated testbed. Second, it presents a specific realization of these principles in the ProtoGENI federation. Third, it shares our experiences building and running this federation, which has been used by more than three hundred experimenters over the last three years. These include experiences that have caused us to re-think parts of the federation's design, an analysis of its robustness to failures, and an evaluation of the time required to set up experiments.

Our five design principles are:

- *Partitioned trust* between the federates: Each federate is responsible for its own resources and users, and only trusts other federates insofar as the peer's resources and users are concerned. Each federate retains the right to make local authorization and policy decisions, and no testbed occupies a privileged position in the federation.
- *Distributed knowledge*: No single entity has complete knowledge of the system. This enables local extensions, allowing federates to offer unique resources and to add new features without being limited by the global framework. It also aids in removing centralized points of failure and inter-facility dependencies.
- *Minimal abstraction*: The ProtoGENI framework provides a low-level API for resource access, rather than hiding the details of resources behind higher-level abstractions. This gives implementers of user tools the flexibility to define their own higher-level abstractions, tailoring them to specific user communities or use cases.
- *Decentralized architecture*: ProtoGENI has only one centralized entity, which is used for bootstrapping and convenience. Operation can proceed without it in most cases. There are no global locks in ProtoGENI; instead, we make use of local transactions to coordinate operations that span federates.
- *Minimal dependencies*: Each ProtoGENI call carries as much context with it as possible. This minimizes dependencies between services, which do not need to contact each other on-line for correct operation.

2 Related Work

Emulab [30] provides a diverse set of experimental resources such as wireless nodes, simulation, and real Internet resources through portals to the PlanetLab [29] and RON testbeds [1]. This control framework is built around a strong assumption of centralized management. There are dozens of testbeds around the world built on Emulab, but until we began work on ProtoGENI, each operated in isolation. The federation of these previously

divided testbeds is the chief user-visible contribution of ProtoGENI, and has required significant architectural changes to the underlying software.

PlanetLab is also a large-scale testbed, distributed around the world. All sites run a common code base, and most maintenance and allocation is done by central entities, called PlanetLab Central (PLC).There are multiple instances of PLC, including one in Europe, another in Japan, and VINI [3], which extends PlanetLab's support for topology control. PlanetLab introduced the idea of "unbundled management," separating user tools from the management of the facility, and we make use of it in ProtoGENI. As part of the GENI project, this federation is evolving along a similar path to the one we present in this paper.

ORBIT [17] and StarBed [14] are built around a centralized use, policy, and maintenance model. ORBIT is a Radio Grid Testbed, providing wireless devices to its users. Due to the restrictions of its physical environment, ORBIT does not "slice" its testbed, but allocates all nodes in its testbed to one experiment at a time [21]. StarBed is specifically designed for virtualization, enabling users to build experimental topologies up to thousands of nodes.

The Open Resource Control Architecture (ORCA) is an extensible architecture which provisions heterogeneous virtual networked systems via secure and distributed management over federated substrate sites and domains [16]. ORCA focuses on mechanisms for providers and consumers (e.g. experimenters) to negotiate access to, configure, and use testbed resources. ORCA provides diverse environments on a pool of networked resources such as virtualized clusters, storage, and network elements which are maintained independently by each site. While ORCA shares many features with ProtoGENI, it uses a different set of fundamental design decisions.

Panlab [27, 28] is a federated testbed with facilities distributed across Europe. While the Panlab and ProtoGENI architectures have many analogous elements, the philosophies behind them differ. Panlab's "Private Virtual Test Labs" (similar to GENI slices) are typically controlled and used through a centralized manager, called Teagle. In contrast, ProtoGENI's architecture emphasizes distributed systems aspects of testbed federation, avoiding centralized services almost entirely.

Namgon Kim et al. [10] have published work on connecting the FIRST@PC testbed with ORCA-BEN. They focus more on the stitching aspects of federation, while we examine the overall architecture and API.

SensLAB is a large scale wireless sensor network testbed, allowing researchers access to hundreds of sensor nodes distributed across multiple sites [24]. The system presents a single portal, through which users can schedule experiments across all the available networks. The current SensLAB installations operate highly homogeneous hardware platforms, but are working toward interoperability with OneLab [2], and we expect that this integration will result in sophisticated federation management facilities.

WISEBED [4] is another distributed system of interconnected testbeds, in which the hardware resources are large scale wireless sensor networks. Like GENI, WISEBED aims to produce a large and well organized structure by combining smaller scale testbeds; the chief difference is that WISEBED focuses on wireless sensor technology, while almost all networked GENI resources use wired links (with a minority of facilities choosing to make wireless resources available for special purposes).

Soner Sevinc [25] has developed an architecture for user authentication and trust in federations using Shibboleth as an identity provider. Our federation architecture provides mechanisms for federates to coordinate experimentation, and has been integrated with Soner Sevinc's system.

Grid [8] and Tier [13] systems share some goals with federated network testbeds, in that they are distributed systems able to connect heterogeneous, geographically distributed resources from multiple administrative domains. Grid systems provide dynamic allocation and management of resources via common tool kits such as Globus [7]. As with our system, each domain is responsible for its own maintenance and policy. The fundamental distinction is that in ProtoGENI, it is the network, rather than computing resources, that is the primary object of interest. Grid computing hides most resource heterogeneity and infrastructure behind abstract interfaces, where as we expose them whenever possible. Cloud computing takes this one step further with virtualization. In comparison, ProtoGENI provides users with more transparent control over the network and the ability to take advantage of the diversity of resources for experimentation. Researchers are investigating the integration of Grid and traditional network testbed resource management techniques [23]; such a combination is largely orthogonal to the peer-to-peer federation model we consider. Most grids are organized hierarchically, while ProtoGENI is decentralized, allowing its principals more autonomy; most clouds consist of resources owned by a single entity, and are thus not federated.

3 Architecture

The architecture of ProtoGENI builds on the GENI framework. In this section, we will describe the overall GENI structure before examining how ProtoGENI expands on that architecture.

3.1 GENI

GENI's architecture is based on the "Slice-based Federation Architecture" (SFA) [19], which has been developed by the GENI community. The SFA is so named because it centers around partitioning the physical facility into "slices," each of which can be running a different network architecture or experiment inside. Physical resources, such as PCs, routers, switches, links, and allocations of wireless spectrum are known as "components;" when a user allocates resources on a component, the set of resources they are given comprises a "sliver." This sliver could be a virtual machine, a VLAN, a virtual circuit, or even the entire component. Each sliver belongs to exactly one slice: in essence, a slice is a container for a set of slivers.

There are two main types of principals in GENI:

Aggregate Managers (AMs) are responsible for managing the resources (components) on which users will create networks and run experiments. AMs are responsible for the allocation of their resources, and may make decisions about who is authorized to use them. An individual AM may manage a collection of components, called an *aggregate*; in practice, each facility in GENI runs a single AM that manages all of its resources, and the largest aggregates contain hundreds of nodes and thousands of links.

Users access components from the federated testbed to run an experiment or a service. A user is free to create slices which span multiple AMs, and each user is authorized by one of the facilities in the federation.

Principals and many other objects in the system are uniquely named by a Uniform Resource Name (URN) [15]. The URN scheme that we use [26] is hierarchical—each authority is given its own namespace, which it can further subdivide if it chooses. To maintain *partitioned trust*, each authority is prohibited, through mechanisms described in [31], from creating URNs outside of its namespace. An example of a GENI URN is:

`urn:publicid:IDN+emulab.net+user+jay`

Because the URN contains the identity of the authority that issued it (in this example "`emulab.net`"), it is possible to tell which authority "owns" the object without resorting to a lookup service; this is in keeping with our *decentralized architecture* goal.

At a high level, testbeds federate in this framework by forming trust relationships: if facility A trusts facility B, then A is willing to trust B's statements about what users it has, what slices they have created, and what resources B offers. Note that this does not preclude A from having local allocation policies: just because it recognizes B's users does not obligate it to satisfy all requests they might make. Arrangements regarding "fair sharing," etc. can be made as part of the federation agreement. Trust relationships need not be symmetric: A may choose to trust B even if that trust is not reciprocated.

3.2 ProtoGENI Architecture

We build on the basic GENI architecture by adding two new kinds of entities into the federation, and by providing an expanded API for AMs.

Slice Authorities (SAs) are responsible for creating slice names and granting users the necessary credentials to manipulate these slices. By issuing a name for a slice, the SA agrees to be responsible for the actions taken within the slice. An SA may be an institution, a research group, a governmental agency, or other organization.

A user has an account with an SA, called the "home" SA; this SA vouches for the identity of the user, and in most cases, is willing to create slices for the user. The user is, however, free to create slices using any SA that, according to its own policies, is willing to be responsible for that user's actions.

Of course, establishing trust in this pairwise fashion does not scale well to large federations. ProtoGENI's sole centralized service, the **Clearinghouse** (CH), is used to make this process more convenient: it allows federates to publish the certificates that are used to establish trust, and to discover the certificates of other federates. It is important to note that this does *not* mandate specific trust relationships: as described in [31], a federate may choose not to trust some certificates stored at the clearinghouse, or may choose to trust additional certificates that are not registered there.

The clearinghouse also serves a second purpose: it acts as a registry where various objects can be looked up. Notably, users can ask the clearinghouse for a list of all registered federates to bootstrap the process of resource discovery, as described in the next section. In both of these roles, the information provided by the clearinghouse changes infrequently, and can be safely cached for long periods of time (days or weeks).

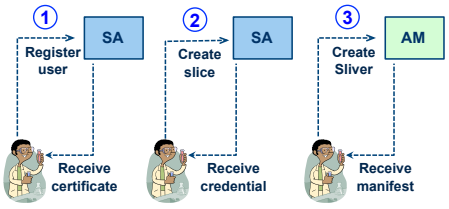

Fig. 1. Overall architecture detailing user interactions with entities in the federation.

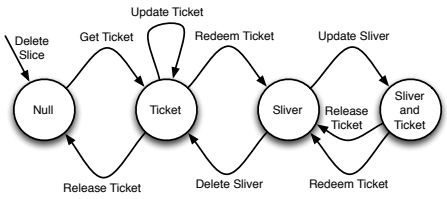

Fig. 2. Life cycle of a sliver. Edges are labeled with the name of the operation that causes the state transition.

ProtoGENI AMs also export an expanded interface relative to the GENI standard. Specifically, they can issue *tickets*, which are guarantees of resource provision for the owner of the ticket. When a user *redeems* one of these tickets with the AM, the AM creates a sliver, and the user can begin running an experiment. ProtoGENI AMs also support the ability to update existing slivers.

ProtoGENI has sophisticated requirements for authentication and authorization, complicated by the fact that different parts of the system are owned and operated by different organizations, each of which may need to enforce custom local policies.

The authentication system is based on the IETF PKIX model [5], while the authorization mechanism involves the presentation of cryptographically signed *credentials* (which behave analogously to X.509 Attribute Certificates [6]). Together, these primitives allow the warranting of identities, the granting and delegation of permissions, and the verification of identity and privilege. Most importantly, all of these operations may be performed by different principals, who need no direct knowledge of each other. ProtoGENI's authentication and authorisation system are detailed elsewhere [31].

3.3 Running an Experiment

Putting these pieces together, a user goes through the following steps to run an experiment (see Figure 1).

Creating a Slice: He contacts an SA (usually the "home" SA), and requests a new slice. If the request is granted, the SA gives him a name for the new slice and the credentials necessary to manipulate it. At this point, no resources are yet allocated.

Discovering Resources: Next, the user selects the components that he will use to build his network. This can be done in a number of different ways. The simplest is to ask each AM in the system to enumerate the resources it has available; the user asks the CH for a list of federated AMs (or uses a cached copy of this list), and then asks each AM for an "advertisement" describing its resources. Alternately, he may make use of network embedding tools [22] to help select appropriate components.

Creating Slivers: Once the user has selected a set of components, he creates a "request" describing the network to be built. The user sends this request to each AM from which he wants resources. When granting such a request, the AM returns a "ticket", promising the use of those resources. If not all ticket requests are granted, the user may

keep the tickets he has and/or try to obtain new tickets to replace the failed requests. Once he is satisfied with the set of tickets held, those tickets may be "redeemed" at the AMs that issued them, causing slivers to be created.

Using Resources: The user may now log into the slivers and begin running experiments. Programming, configuring, or logging in to a sliver is done with component-specific methods; many components are accessed through `ssh`, but other access methods may be required for some components. The user may modify the slice while it is running, and releases all slivers when the experiment is complete.

4 Interactions Between Federates

ProtoGENI has been designed to keep federates as loosely coupled as possible; they do not depend on central services, and the only parts of the system involved in a given operation are those directly affected by it. In the extreme case, if a federate is cut off from communication with the rest of the federation, users who can reach the federate are still able to create slices and slivers on it.

This is possible because, in keeping with the design principles of *decentralized architecture*, *minimization of dependencies*, and *distributed knowledge*, ProtoGENI goes to great lengths to ensure that *minimal state synchronization* is required between AMs, SAs, and the CH. This section describes the interactions these services have with each other and with users. We concentrate on where ProtoGENI stores state, how it avoids centralized services, and how failures are managed. Because the full ProtoGENI APIs [20] are too large to cover in depth here, we introduce only the calls necessary to understand state management.

4.1 Slice State

ProtoGENI does not attempt to guarantee a globally consistent view of the state of each slice. Instead, it uses a loose consistency model in which each of the individual authorities and managers maintain their own state.

The authoritative source for user and slice records is the SA that issued them, and the authoritative source for sliver information is the AM on which the sliver exists. Because the URNs used in ProtoGENI encode the issuing authority, it is possible to determine the correct authority to query simply by examining an object's name. If, for example, a AM wishes to find out more about a user who has created a slice on it, the AM may use the user's URN to identify and query the user's home SA.

When a sliver is created, the AM is *not* provided with a global picture of the slice: the sliver request (whose format is covered in Section 5) need only contain the resources on the AM in question and any links to directly adjacent AMs that need to be configured as part of the slice. Information about the rest of the slice is not needed for the AM to create its slivers, and maintaining a global view would require that the AM be notified of changes *anywhere* in the slice, even if those changes do not directly affect it.

As a convenience, SAs register users and their slices at the CH. Records at the CH are not considered authoritative, however, since a network partition might delay updates to

them. Nor does the CH maintain a list of slivers; this list is constantly changing, and could never be completely up to date without adding a large amount of synchronization (and consequently delay) to the system. Each AM attempts to inform a slice's SA whenever a sliver is created or destroyed, but as with records in the CH, these data are advisory rather than authoritative.

Slice and Sliver Lifetimes Because authoritative slice state is distributed across SAs and AMs, and we cannot guarantee that they remain in contact throughout the lifetime of the slice, we give each slice and sliver an expiration date. This way, we can be assured that all slivers are eventually cleaned up and their resources reclaimed.

There are important nuances, however, in the relationship between slice and sliver lifetimes. Because each sliver must belong to a slice, the sliver must not outlive its slice. If it did, this could lead to a situation in which the user would lose control of the sliver.

The first consequence of this requirement is straightforward: the expiration time for each sliver is bounded by the expiration time of the slice itself. The slice credential that is generated by an SA when then slice is created contains that slice's expiration time. When slivers are added to the slice, AMs must simply ensure that the slivers' expirations are no later than the slice's expiration.

The second consequence is that a slice cannot be deleted before it expires. It is possible that slivers exist that the SA is unaware of; a AM may have been unable to contact the SA to inform it of the sliver's existence. Therefore, the SA cannot know with certainty that deleting the slice is safe and will leave no orphaned slivers. As a result, slice names cannot be re-used by experimenters before they expire. Since the namespace for slices is effectively unbounded in size, this is not a major concern.

Both slices and slivers may be renewed before they expire; the slice's lifetime must be extended before the slivers'.

Resource Reservation Across AMs Slices that cross AMs present a dilemma: we would ideally like the process of allocating or updating slivers to be atomic across all AMs. As a concrete example, consider a slice with existing slivers from two different AMs. We would like to make a change on both slivers, but only if both of the changes succeed. If either one is denied, we want to roll back to the original configuration without losing existing resources or otherwise changing the slivers. However, the loosely-coupled nature of the federation precludes using global locks or global transactions.

Instead, we consider the resource allocation process on each AM to be a separate local transaction, and model the life cycle of each sliver as a state machine, shown in Figure 2. We designed the state machine with *minimal abstraction* in mind, allowing clients or other intermediaries to build a transactional abstraction across AMs on top of our lower-level per-AM API. Each sliver can be in one of four states:

1. The *Null* state, in which the sliver does not exist (has not yet been created, or has been destroyed).
2. The *Ticket* state, in which the user holds a ticket promising resources, but the sliver is not instantiated on the component.
3. The *Sliver* state, in which the sliver has been instantiated, but the user does not hold a valid ticket.

4. The *Sliver and Ticket* state, in which the user has both an instantiated sliver and a ticket.

This state machine makes sliver manipulation a three-step process:

1. Get the list of currently available resources from each AM.
2. Request a new ticket on each AM; this step obtains a *guarantee* of resources, but does does not actually instantiate a new sliver or modify an existing sliver.
3. Redeem the tickets at each AM to "commit" the resource change.

Steps 1 and 2 are not atomic: if other users are simultaneously trying to reserve resources to their own slices, the second step may fail. In a distributed system like ProtoGENI, it is not feasible to *lock* the resource lists for any length of time. Since contention for resources is generally rare in ProtoGENI, a form of *optimistic concurrency control* [11] is employed to both avoid locking and to ensure that users will find out if someone else has already reserved a resource.

If the second step fails on some AMs, but not others, the user has three options. First, he can decide to simply redeem the tickets that he *was* successful in getting. A user trying to get as many slivers as possible might employ this strategy. Second, he can abort the transaction by releasing the new tickets he obtained. This will return the slivers to their previous states (either *Null* or *Sliver*) without modifying them. Third, he can employ a more sophisticated strategy, in which he holds onto the tickets that he did receive, and requests tickets from a *new* set of AMs to replace those that were denied.

Our experience running the Emulab testbed [30] suggests that retries due to the race between steps 1 and 2 will be rare. Emulab uses a similar optimistic model in which the resource database is not locked during allocation, and despite the fact that Emulab sees heavy use, of 9,500 experiment swap-ins (analogous to ProtoGENI sliver creations) in the past year, only 21, or 0.2%, had to be retried due to contention for resources.

In addition to its lifetime, each ticket has a "redeem by" time, which is set to expire much sooner; typically, in a matter of minutes. If the user does not redeem the ticket in time, the resources are reclaimed. This guards against clients that do not complete their transactions in a timely fashion.

4.2 Behavior in the Face of Failures

ProtoGENI passes as much context as possible in API calls, so that they can be *self-contained*. While this does result in some extra overhead in the calls, the benefit is that the user can continue to make progress in the presence of network or service failures. For example, a user obtains authorisation credentials from his home SA, and these credentials are passed *by the user* to AMs when requesting tickets. As described in [31], the AM receiving this material can verify its authenticity without contacting the issuer.

The result is that users can continue to use the system in the face of failures in one or more services, including the CH. For example, if an SA is down, its users cannot create new slice names, but can continue to interact with any existing slices and slivers. As long as they do not lose the slice credentials obtained upon slice creation, there is no need to contact the SA to manipulate the slivers in the slice. The lone exception is to *extend* the life of a slice before it expires. However, this can be done at any time before the slice expires, so transient errors are not fatal.

While SAs attempt to register new slices at the CH, and AMs attempt to register new slivers with the slice's SA, failure to do so does not cause slice or sliver creation to fail. Making this registration mandatory would significantly increase the dependencies in the system, and reduce its ability to operate in the face of service failure.

5 Resource Specification

Resource specification is a core part of interacting with a testbed and must fulfill three functions: First, AMs must be able to describe their available resources. Second, users need to describe what resources they would like to acquire. Third, AMs must provide information required for users to make use of those resources.

To perform these functions, we have developed a new resource specification format, RSpec, which is an XML-formatted descriptive language. In keeping with the principle of *minimal abstraction*, our specification is declarative rather than imperative. While an imperative language would add descriptive power, it is more difficult to analyze and manipulate. Adopting a descriptive format makes it possible for many tools to process and transform resource descriptions, and encourages composition of tools.

One key principle behind our RSpec design is *distributed knowledge*. Because knowledge of resources is distributed, every entity in the system can independently provide information about resources. We use progressive annotation to allow a client to coordinate data from multiple sources: operations take a resource specification as input and yield that same specification annotated with additional information as output.

5.1 Annotation

Our specification format comes in three variants, each one designed to serve a slightly different function. Multiple entities in the system can provide knowledge about a single resource, and we allow calls to these entities to be composed by a client. An RSpec describes a topology made up of nodes, interfaces, and links that connect them. Annotations can provide additional information about the topology or resources.

Advertisements Advertisements are used to describe the resources available on a AM. They contain information used by clients to choose components. The AM provides at least a minimal description of these resources. Our architecture then allows its advertisement to be passed to measurement services or others who may discover more information about the resources. At each such service, the advertisement is further annotated.

This progressive annotation of advertisements enables our *distributed knowledge* model. An AM can change the set of resources advertised without first notifying or negotiating with other federates. AMs provide their own authoritative information about which resources they manage and the availability of those resources.

At the same time, other entities can describe resources. A service might measure shared resources to determine which ones are the most heavily used. Or it might provide information about bandwidth and delay between networked components. Annotating services do not need to coordinate with any AMs, who need not even be aware of their existence.

Requests Requests specify which resources a client is selecting from AMs. They contain a mapping between physical components and abstract nodes and links.

When the client has an advertisement with the information they need, they create a request describing which resources they require. Some requests, called bound requests, specify precisely which resources are needed—"I want `pc42`, `pc81`, and `pc9`." Other requests provide a more abstract, or unbound, description—"I want any three PCs."

Once the request is generated, it goes through an annotation process similar to that of the advertisement. If there are unbound resources in a request, it may be sent along with an advertisement to an embedding service which annotates the request with specific resource bindings. In order to *minimize dependencies*, embedding services are not associated with a particular AM and receive both a request and one or more advertisements from the client. The client then sends the bound request returned by the embedding service to a AM in order to acquire the resources. Each resource is tagged with the AM that should allocate it. This means that the client can simply send the same request to all relevant AMs and each one will allocate only those resources which it controls.

Manifests Manifests provide information about slivers after they have been created by the AM. This information is typically needed by users so that they can make practical use of the resources: details such as host names, MAC addresses, `ssh` port numbers, and other useful data. This information may not be known until the sliver is actually created (e.g. dynamically assigned IP addresses). The manifest is returned to the user upon sliver creation.

5.2 Extensibility

To allow AMs to make unique resources available, we must provide a mechanism for allowing them to advertise new kinds of resources. Our core RSpec is therefore designed for *distributed knowledge*, allowing different federates to provide their own independent extensions. The base RSpec schema verifies within a single namespace and allows elements or attributes from other namespaces to be inserted anywhere. These fragments are then verified using extension-specific schemata. We allow extensions on any variant of the RSpec, thus allowing extensions to be created and used by any entity in the federation.

Our extensions have a number of useful properties:

1. *Extensions are safely ignored:* Not all clients or servers will support all extensions. If a client or server does not support a particular extension, then the tags which are part of that extension will be ignored. This allows extensions to be created and deployed incrementally with much greater ease than a change to the core RSpec.
2. *Extensions are modular:* Each extension can mix elements or attributes into the main RSpec, but those elements and attributes are explicitly tied to the extension namespace. Every extension can co-exist with every other extension.
3. *Extensions are validated:* In order to tag extensions, each one uses a unique XML namespace. We are thus able to validate any XML document using the core schema. The extensions themselves are also validated. We use independent schemata for each

extension and validate the elements in the namespaces for each extension against its schema.

6 Experiences

The primary indicator of the success of our design is that ProtoGENI is a running, active system; the current federation contains sixteen AMs, and over 300 users have created more than 3000 slices. To evaluate our system more concretely, we first describe our experiences in creating and running the ProtoGENI federation, then show results from quantitative tests of the system.

6.1 Framework Design

ProtoGENI has been open to users and tool developers throughout its development. This allowed our experiences with actual users and experimenters to guide our design decisions. Described below are some of the lessons we have learned from seeing our system used by others.

Slice and Sliver Lifetimes: One area of the system that required very careful design was the lifetime of slices and slivers. We have found that this aspect of the system is consistently confusing to users; they expect to be able to delete *both* slivers and slices, and have trouble understanding why slice names cannot be deleted before they expire. However, as discussed in Section 4.1, this cannot be allowed, given the decentralized architecture of our system: an SA cannot be sure that all slivers are really gone until the expiration time on the slice (which bounds the expiration time on its slivers.)

Adding to the confusion is the fact that a slice name can be *reused* on an individual AM. In other words, the holder of a slice credential may create, destroy and then create a sliver again. As far as the AM is concerned, if no local sliver currently exists for a slice, then it is willing to create one. In fact, this is exactly what many users do.

Users will often create a slice name, and then use that credential to create and destroy many slivers on the same AM. This works since users usually know the state of their own experiments. It has also resulted in an unintended consequence: users may create a slice name, and set the expiration time to many months in the future. Since users often forget to destroy their slivers, resources can get tied up doing nothing for a long time. When this became a common problem, we established a policy which requires slices to be renewed every five days.

UUIDs: Our initial strategy was to use UUIDs [12] as identifiers. One advantage was that they can be generated by any party with a high confidence that they will be unique. They are also opaque, meaning that clients do not have to do any work in parsing or interpreting them. However, we discovered that using a flat namespace for all objects had one major drawback.

There is no inherent mapping between identifiers and authorities. A flat namespace requires a lookup service to resolve the authority issuing an identifier. For example, verifying that an identifier was issued by a particular authority would require one to first resolve the UUID to that authority. While decentralized resolvers for flat namespaces

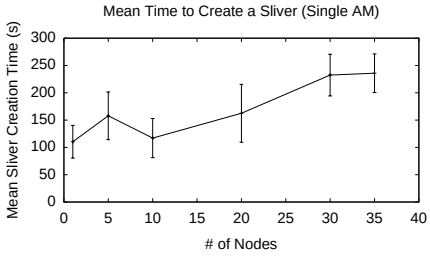

Fig. 3. Mean time to create slivers of various sizes

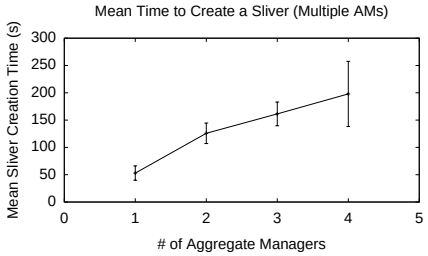

Fig. 4. Mean time to create slivers on multiple AMs

do exist (such as DHTs), we saw that including the authority in the identifier, and thus skipping this first step, was more in keeping with our minimization of dependencies principle, so that operations require contacting only the entities directly involved in them.

Sliver Lifecycle: When designing the ProtoGENI API, we tried to make it easy to use resources that are distributed geographically, under *different administrative domains*, and controlled by distinct management authorities. Users are faced not only with the task of deciding what resources they want, but they must also contact independent authorities to ask for those resources. As described in Section 4.1, dealing with resource allocation failures is complicated for both the system and the users. Worse still are sliver updates, especially those that span multiple AMs.

The *life cycle* diagram shown in Figure 2 was the result of user experience and multiple design iterations. An early version of the API used the same method to modify both an unredeemed ticket and an existing sliver. In the latter case, the user had to present the original ticket, even though the ticket had already been redeemed and was technically worthless. The user was also required to hang onto this ticket in case he wanted later modify the sliver. Furthermore, if the user decided to release this new ticket, he was left in an even less complete state, with an active sliver and no ticket to modify it later. As described in Section 4.1, this sequence is a common activity, as users allocate and modify resources across a set of AMs.

As more users signed up to run experiments with ProtoGENI, we received numerous questions about updating tickets and slivers. It was at this point that we decided to formalize the sliver lifecycle as a state machine, but our early attempts resulted in designs with a large number of states, making them difficult to understand. We finally arrived at the state machine in Figure 2, which puts the user in control of what to do with denied ticket requests, while minimising the size of the state machine.

6.2 Testing the System

In this section, we look at two metrics. We measured the time required to create slices both on a single AM and across multiple federates. In addition, we injected various faults and examined the behavior of the system when dealing with them.

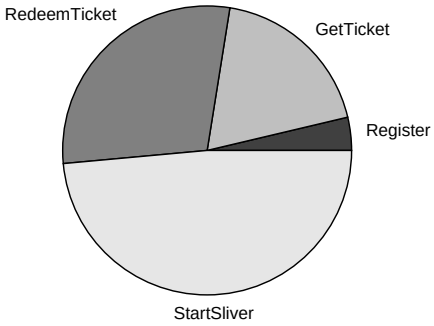

Fig. 5. Breakdown of time spent creating sliver

Table 1. Operability during failures. Each row is an attempted operation. Columns are component failures. Cells show if the given operation succeeds, *can* succeed if the client has cached the indicated object, or always fails (×).

	Failed Entity		
Operation	CH	SA	AM
Discover Res.	*AM List*	*Self Cred.*	×
Create Slice	Success	×	Success
Get Ticket	Success	*Slice Cred.*	×
Redeem Ticket	Success	*Slice Cred.*	×
Start Sliver	Success	*Slice Cred.*	×
Stop Sliver	Success	*Slice Cred.*	×
Delete Sliver	Success	*Slice Cred.*	×
Sliver Login	Success	Success	Success

Slice Creation Time We have run tests on the federation to determine the duration of typical user tasks. For our experiment, we ran a test script using the following sequence of steps: get a user credential, create a slice, list component managers at CH, discover resources on one or more AMs, get a ticket, and finally redeem the ticket. We ran these tests with up to 35 nodes to see how increasing the size of the request affected the results. Figure 3 shows our results.

We also ran experiments using multiple AMs: between one and four. Figure 4 shows the time required to allocate nodes as the number of AMs increases. Each trial allocated 20 nodes total, but split the allocation of those nodes across a different number of AMs. As the number of AMs used increases, so does the time required to allocate nodes. This increase in time could be mitigated by contacting the AMs in parallel. The single AM case was run at a relatively lightly-loaded AM and so runs unusually fast relative to the times seen in Figure 3.

The time spent creating a ten node sliver (averaged over multiple runs) is detailed in Figure 5. The *Register* step accounts for all the negotiation with the SA to allocate the slice name and obtain slice credentials. All subsequent steps are carried out at the AM: *GetTicket* reserves components to the sliver, and is potentially expensive because it attempts sophisticated optimization not only to satisfy the immediate request, but also to maximize the proportion of *future* sliver demands which can be met [22].

The following *RedeemTicket* stage prepares the allocated components for use in the sliver: for instance, disk images are transferred [9] to nodes where necessary, and VLANs are programmed into switches (but not brought up). Our current implementation also performs auxiliary tasks for user convenience at this point (such as configuring DHCP servers with information about control network interfaces, and registering DNS names for nodes in the sliver). The final *StartSliver* period is frequently the lengthiest, although much of the operation is beyond the direct control of the AM. It involves rebooting each node in the sliver into the newly defined environment, as well as completing configuration tasks that are most easily deferred until boot time (such as configuring routing tables for the experimental networks).

Some of the time a client spends interacting with the system is spent gathering information it has already fetched during previous runs, such as the list of AMs, the user's "self credential," and credentials for slices that are being reused. In addition to providing additional robustness in the face of failure, caching these values can reduce the time it takes for a user to successfully create slivers. This provides a constant time speedup regardless of how many nodes or AMs are involved in creating a sliver. Our experiments show that this reduces the time taken to create a sliver by 17 seconds on average.

Behavior in the Face of Faults Our federation was designed to cope with failure of one or more elements. In order to test this, we injected faults into the system in order to emulate a network partition. In each test, one federated entity was isolated from the federation and the client. We then attempted various operations to see whether they succeeded, failed, or required some cached client-side information to work. The results of these experiments are shown in Table 1.

If the CH fails or is isolated from the federation, the client can still perform operations on any AM it can reach, as well as the client's SA. In order to discover resources on a AM, the client would need to have a cached list of AMs in the federation; because this list does not change frequently, most clients will have a fairly recent copy. The only consequence of using an out-of date AM list is that the client may miss the opportunity to use AMs that have recently joined.

The SAs are responsible for managing users and their slices. If an SA fails, its users can no longer create slices or acquire credentials to manipulate existing slices. If a user already has a self credential, it can be invoked to discover resources on any AM in the federation. If the user has already created a slice, it has not expired, and the client has cached the slice credential, then the user can still create and manipulate slivers on any AM.

When a AM fails, the user cannot perform resource discovery or sliver creation, or any sliver manipulation calls on that *particular* AM. Depending on whether the failure is with the AM or the component itself, the user may still be able to log in to the slivers that have already been created. The failure of one AM does not affect the user's ability to use other AMs in the federation.

7 Conclusion

Federated testbeds provide new opportunities for experimentation, but also raise a number of design challenges. We applied five design principles to the design of ProtoGENI, resulting in a loosely coupled system that preserves local autonomy for federates. ProtoGENI provides a low-level interface to discovering and reserving testbed resources; our future work will build upon this fundamental framework to provide higher-level abstractions and services for experimenters.

Acknowledgments

Many people have been active participants in the GENI design process, which arrived at the basic design described in Section 3. While the total number of contributors to this process is large, we would like to specifically acknowledge the chairs of the GENI Facility Architecture Working Group and heads of the GENI control frameworks: Larry Peterson (PlanetLab), John Wroclawski (TIED), Jeff Chase (ORCA/BEN), and Max Ott (OMF). Others major contributors to the design process have included Aaron Falk, Ted Faber, Steve Schwab, and Ilia Baldine.

References

1. Anderson, D.G., Balakrishnan, H., Kaashoek, M.F., Morris, R.: Resilient overlay networks. In: Proc. of the ACM Symposium on Operating Systems Principles (SOSP). Banff, Canada (Oct 2001)
2. Antoniadis, P., Fdida, S., Friedman, T., Misra, V.: Federation of virtualized infrastructures: sharing the value of diversity. In: Proc. of the 6th International Conf. on emerging Networking EXperiments and Technologies (CoNEXT). Philadelphia, PA (Nov 2010)
3. Bavier, A., Feamster, N., Huang, M., Rexford, J., Peterson, L.: In VINI veritas: Realistic and controlled network experimentation. In: Proc. of the ACM Special Interest Group on Data Communication (SIGCOMM). Pisa, Italy (Aug 2006)
4. Chatzigiannakis, I., Koninis, C., Mylonas, G., Fischer, S., Pfisterer, D.: WISEBED: an open large-scale wireless sensor network testbed. In: Proc. of the 1st International Conf. on Sensor Networks Applications, Experimentation and Logistics (Sep 2009)
5. Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W.: Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Request for Comments 5280, IETF (May 2008)
6. Farrell, S., Housley, R., Turner, S.: An internet attribute certificate profile for authorization. Request for Comments 5755, Internet Engineering Task Force (Jan 2010)
7. Foster, I., Kesselman, C.: Globus: A metacomputing infrastructure toolkit. International Journal of Supercomputer Applications 11(2) (Summer 1997)
8. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: Enabling scalable virtual organizations. International Journal of High Performance Computing Applications 15(3) (Aug 2001)
9. Hibler, M., Stoller, L., Lepreau, J., Ricci, R., Barb, C.: Fast, scalable disk imaging with Frisbee. In: Proc. of the 2003 USENIX Annual Technical Conf. pp. 283–296. San Antonio, TX (Jun 2003)
10. Kim, N., Kim, J., Heermann, C., Baldine, I.: Interconnecting international network substrates for networking experiments. In: Proc. of the 7th International ICST Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom). Shanghai, China (Apr 2011)
11. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Transactions on Database Systems 6(2) (Jun 1981)
12. Leach, P., Mealling, M., Salz, R.: A universally unique identifier (UUID) URN namespace. Request for Comments 4122, Internet Engineering Task Force (Jul 2005)
13. McKee, S.: The ATLAS computing model: status, plans and future possibilities. Computer Physics Communications 177(1–2) (Jul 2007)

14. Miyachi, T., Basuki, A., Mikawa, S., Miwa, S., Chinen, K., Shinoda, Y.: Educational environment on StarBED: case study of SOI Asia 2008 spring global E-Workshop. In: ACM Asian Conf. on Internet Engineering. Bangkok, Thailand (Nov 2008)
15. Moats, R.: URN syntax. Request for Comments 2141, Internet Engineering Task Force (May 1997)
16. The ORCA GENI control framework. `http://www.nicl.cs.duke.edu/orca`
17. Ott, M., Seskar, I., Siraccusa, R., Singh, M.: ORBIT testbed software architecture: Supporting experiments as a service. In: Proc. of the International ICST Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom). Trento, Italy (Feb 2005)
18. Peterson, L., Bavier, A., Fiuczynski, M.E., Muir, S.: Experiences building PlanetLab. In: Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI). Seattle, WA (Nov 2006)
19. Peterson, L., Ricci, R., Falk, A., Chase, J.: Slice-based federation architecture. `http://groups.geni.net/geni/wiki/SliceFedArch` (Jun 2010)
20. ProtoGENI Project: ProtoGENI API. `http://www.protogeni.net/trac/protogeni/wiki/API` (May 2012)
21. Rakotoarivelo, T., Jourjon, G., Ott, M., Seskar, I.: OMF: A control and management framework for networking testbeds. ACM SIGOPS Operating Systems Review 43(4) (Jan 2010)
22. Ricci, R., Alfeld, C., Lepreau, J.: A solver for the network testbed mapping problem. ACM SIGCOMM Computer Communication Review (CCR) 33(2), 65–81 (Apr 2003)
23. Ripeanu, M., Bowman, M., Chase, J.S., Foster, I., Milenkovic, M.: Globus and PlanetLab resource management solutions compared. In: Proc. of the 13th IEEE International Symposium on High-Performance Distributed Computing (HPDC '04). Honolulu, HI (Jun 2004)
24. des Roziers, C.B., Chelius, G., Ducrocq, T., Fleury, E., Fraboulet, A., Gallais, A., Mitton, N., Noël, T., Vandaele, J.: Using SensLAB as a first class scientific tool for large scale wireless sensor network experiments. In: Proc. of the 10th International IFIP TC 6 Conf. on Networking (NETWORKING '11). Valencia, Spain (May 2011)
25. Sevinc, S.: A path to evolve to federation of testbeds. In: Proc. of the 7th International ICST Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom). Shanghai, China (Apr 2011)
26. Viecco, C.: Use of URNs as GENI identifiers. `http://gmoc.grnoc.iu.edu/gmoc/file-bin/urn-proposal3.pdf` (Jun 2009)
27. Wahle, S., Tranoris, C., Denazis, S., Gavras, A., Koutsopoulos, K., Magedanz, T., Tompros, S.: Emerging testing trends and the Panlab enabling infrastructure. IEEE Communications Magazine 49(3) (Mar 2011)
28. Wahle, S., Magedanz, T., , Campowsky, K.: Interoperability in heterogeneous resource federations. In: Proc. of the 6th International ICST Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom). Berlin, Germany (May 2010)
29. Webb, K., Hibler, M., Ricci, R., Clements, A., Lepreau, J.: Implementing the Emulab-PlanetLab portal: Experience and lessons learned. In: Proc. of the First Workshop on Real, Large Distributed Systems (USENIX WORLDS). San Francisco, CA (Dec 2004)
30. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI). Boston, MA (Dec 2002)
31. Wong, G., Ricci, R., Duerig, J., Stoller, L., Chikkulapelly, S., Seok, W.: Partitioning trust in network testbeds. In: Proc. of the 45th Hawaii International Conf. on System Sciences (HICSS-45). Wailea, HI (Jan 2012)