Preliminary Review of the GENIAPI as Control Framework Interoperability Architecture and TIED Federation Plug-In Candidate

Ted Faber, USC/ISI

John Wroclawski, USC/ISI

Version 1.2

August 12, 2010

Corresponding to GENIAPI version 0.9, as presented at the 8th GENI Engineering Conference (GEC8), July 20-22, 2010.

Table of Contents

1 Introduction

This document assesses the emerging GENIAPI standards and implementation as a potential target plug-in API for the TIED/DETER[1] control architecture, and as a general GENI interoperability framework. This version 1.2 of the document corresponds to GENIAPI version 0.9, as outlined at the 8th GENI Engineering Conference (GEC8), July 20-22 2010.

We find that the current GENIAPI standard is difficult to adopt as a TIED plug-in target due to its preliminary nature. At the current state of its development, we conclude that using it would not result in a more portable plug-in, nor would it reduce developer workload when accessing different control frameworks. We conclude that the standard is similarly underdeveloped as an interoperability vehicle at this time. In both cases, however, we conclude that if efforts are made to further standardize essential GENIAPI functions and features it can meet the desired goals.

The TIED federation framework has been used successfully to construct experimental environments that span multiple dissimilar facilities. As one part of this process, a federated experiment environment is collaboratively constructed and then instantiated. In the instantiation phase, each facility maps the necessary control actions, resource access permissions and principal identities into local actions, access controls and identities. Facilities do this by implementing the required mapping functions and interfaces in the form of a *plug-in*, which is written to a standard specification[1]. TIED/DETER currently supports plug-ins, and thus can create federations across, the DETER testbed, Emulab[2] systems, ProtoGENI[3], deterministic-resource networks controlled by DRAGON[4], and a number of other resource categories.

This federation system has developed in parallel with the Slice-based Federation Architecture[2] (SFA) [5] originally outlined by members of the network testbed community under the auspices of the GENI Planning Group, and further developed by the community under the umbrella of the GENI Program Office and the GENI program.

The emerging GENIAPI[6] is a derivation and standardization of concepts from the SFA and the existing implementations of these concepts in the individual GENI control frameworks, notably the ProtoGENI[3] and PlanetLab[7] efforts. The goal of the GENIAPI is to standardize an API for interoperability between existing and future control frameworks, components, and aggregates. Practically, the design and standardization aspects are balanced against documenting what the initial implementations have done and incorporating their insights. The GENIAPI aggregate manager interface, which we describe in more detail below, has been implemented to varying degrees in PlanetLab, ProtoGENI, and OpenFlow[8][9].

---

[1]The TIED/DETER control architecture is an evolving testbed control and federation architecture developed under the dual auspices of the DETER Cybersecurity Testbed project and the TIED GENI project. We occasionally refer to this architecture as either the "TIED" or the "DETER" architecture in cases where the dual TIED/DETER formulation is excessively awkward and not needed for clarity.

[2]Initially referred to as the "Slice-based Facility Architecture".

The purpose of this document is to describe our review of the GENIAPI version of July, 2010 as a target facility interface for the TIED federation architecture, and more generally as a vehicle for creating points of interoperability between GENI components and control frameworks. It is important to understand that the GENIAPI is at a very early point in its development, and that the observations and conclusions of this document are dependent on the current state of the GENIAPI. One objective of the document is to call out specific areas where progress in developing the GENIAPI would significantly improve its usefulness as a TIED plug-in target and as an interoperability mechanism.

1.1 Interrelating Control Frameworks and Aggregates

In order to understand and evaluate the GENIAPI we must understand how it relates to the SFA and GENI interfaces as a whole. The GENIAPI is an interface between a *control framework*, which implements the slice abstraction, and resource *aggregates* that control federated resources that share an owner. *Slices* are collections of resources that the control framework acquires from resource aggregates. A collection of resources allocated within a single aggregate is called a *sliver*. At the request of a researcher, a control framework will allocate a slice and populate it with slivers from one or more aggregates. A primary goal of the GENIAPI is that aggregates can interoperate with multiple control frameworks[10].

The GENIAPI and the SFA document both concentrate on the definition of the aggregate interface[3], which is the interface between the control framework and the resource aggregate. The interface between users and the control framework is not specified leaving two possibilities for the control framework interface: either there is an unspecified interface to control framework functions such as creating slices, or the control frameworks and aggregates export the same interface recursively.

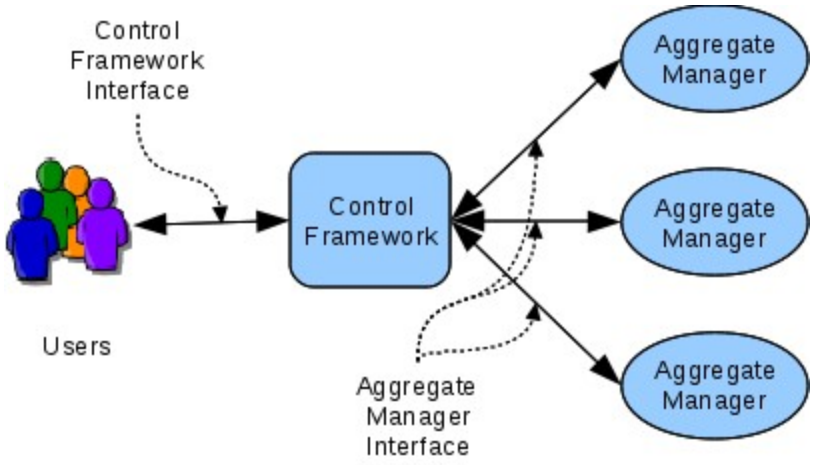

Figure 1: Two-Level Control Framework Interface

Figure 1 shows the case where the control framework has a distinct interface, which we call the two-level model. A control framework implements the slice abstraction internally and collects resources into slices on behalf of users using delegated credentials. Aggregate managers are unaware of slices, they just understand slivers. Aggregates can export slivers to multiple control frameworks.

[3]The SFA document calls this the slice interface.

The recursive model is shown in Figure 2. In the recursive model, the interface between a user and a control framework and the interface between control framework and an aggregate are identical. Slices and slivers are much more similar in this model. The slice a control frame work collects for a user is a collection of slivers from aggregate managers. Because the interface is recursive, those slivers may in turn be made up of slivers from other managers exporting the same interface. Aggregates can still export resources to different control frameworks, but each aggregate may also act as a control framework.

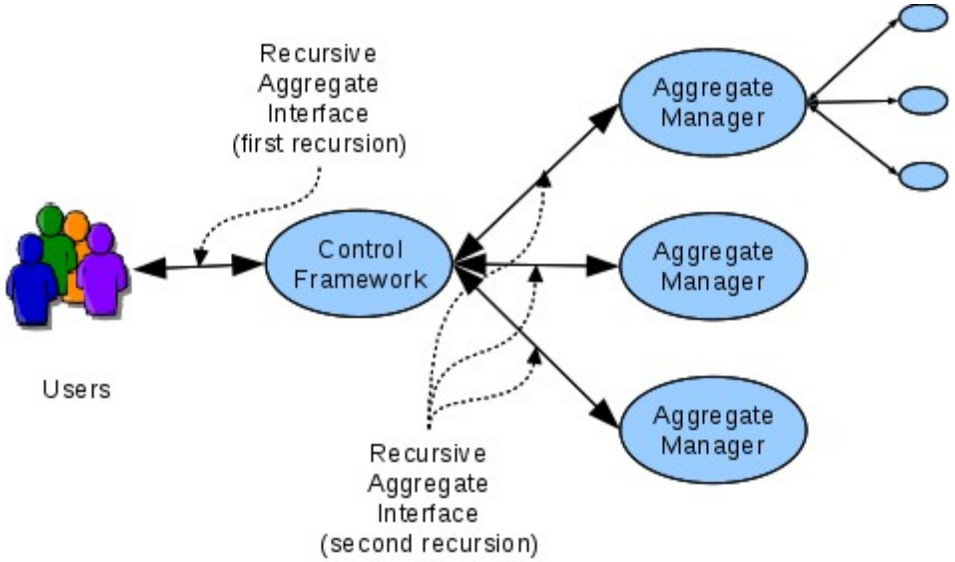

Figure 2: Recursive Control Framework Interface

The two-level model makes the distinction between control framework and aggregate manager clear and separates the concerns of managing a slice abstraction and managing and reserving resources. The recursive model is attractive in naturally supporting multi-level allocation frameworks.

After reviewing the SFA, existing implementations, and the GENIAPI 0.9 specification, it remains somewhat unclear which of these is intended. The SFA document and ProtoGENI tutorial[11] both describe the aggregate manager calls as being made by users. One may construe a “user call” as “a call on the user's behalf by a control framework” in the SFA document, but the ProtoGENI tutorial indicates that the developers intend real user/aggregate interactions. The tutorial also describes users addressing multiple aggregate managers. This indicates that the ProtoGENI developers expect the aggregate interface to be visible to users as in the recursive model, though they neither delineate nor advocate that model.

PlanetLab's SFA model[12] looks more like the two-level model in that users see one facility that binds resources from different administrative domains into one slice. The names of the calls are similar to those used in the aggregate interface, but the semantics differ sufficiently that the developers appear to intend a distinct control framework interface.

The confusion about terminology and usage implies that there may be confusion in the GENI community about the role of the aggregate manager interface. For the remainder of this document we will take the GENIAPI interface to be the control framework to aggregate interface used within a two-level interoperability framework.

1.2 Evaluation Summary

Our top level finding is that the GENIAPI is not at present a viable plug-in target because the interface does not standardize enough of the resource allocation process. Ideally a plug-in written to the GENIAPI would be able to access resources from a variety of control frameworks, but the current state precludes that design.

The GENIAPI calls are sufficient for allocating resources once the plug-in has credentials and a slice in hand, but the GENIAPI does not address how to acquire these necessary elements. In practice, different control frameworks have different credentials and slice representations, with different requirements and operations to acquire and manipulate them. Similarly, while the GENIAPI provides a mechanism for delivering a resource request to the control framework, the format of that request remains different for each framework. Much of the process of allocating resources from a control framework remains unspecified.[4]

Specific operations and concepts that the GENIAPI *does* standardize that are of importance to this conclusion are

- Basic resource allocation
- Separation of credentials and naming
- Soft-state management of slices and slice expiration
- Sliver shutdown

Specific operations and concepts that the GENIAPI does not standardize that are relevant to this conclusion are:

- Slice creation or slice/sliver interaction
- Credential and access control formats and bindings
- Resource request formats

To clarify the first bullet in the second list above: in our description of the two-level model we pointed out that in principle, aggregate managers could operate without knowledge of slices, but as the API is specified and implemented this is not the case. The GENIAPI 0.9 requires a slice in order to create slivers, as do ProtoGENI and PlanetLab implementations. If that constraint were removed from the interface, a TIED plug-in could implement the GENI slice abstraction internally, placing itself in the position of a control framework relative to the aggregate. Conversely, if the GENIAPI supported a standard slice creation, as it would under the recursive model, the plug-in could request GENI slices from the GENIAPI just an any other user. We discuss the issues of slices in sliver creation further in Section 5.3.

[4]That said, we intend to update the TIED ProtoGENI plug-in to access ProtoGENI through the GENIAPI interfaces in preference to the ProtoGENI interfaces to simplify the eventual transition to a complete GENIAPI. As we discuss below, the plug-in will make slightly different use of these interfaces depending on the consensus GENIAPI model.

We also consider the GENIAPI as a framework for exporting resources to multiple control frameworks. We identify several places where the interface needs to become more solid in order to provide enough standardization on which to code and discuss them in following sections. These areas are, broadly:

- Identity definition
- Format and semantics of credentials
- Slice definition
- Resource specification

The rest of this document describes TIED and the existing GENIAPI effort, expands on the specific areas in which we find that the GENIAPI is currently underdeveloped as the basis of a multi-framework TIED plug-in, and outlines what we think are the most important next steps for the GENI community to take in standardizing the API. Section 2 reviews the TIED system, Section 3 describes the current state of the SFA document and GENIAPI specification, Section 4 presents in more detail our findings regarding the current issues and limitations of GENIAPI, and Section 5 describes what we think are the essential missing pieces in the GENIAPI and the next steps to addressing those shortcomings.

2 TIED Federation and Plug-Ins

We briefly outline the architecture and goals of the DETER Control Architecture (DCA). The design document for the ProtoGENI plug-in[13] discusses this in more detail.

The DETER Control Architecture, on which DETER and TIED are based, supports as its basic abstraction a substantial generalization of the Utah Emulab[2] model of experiments and experiment creation. Some key properties of this model are as follows:

- Experiments consist of logical *nodes*, which may model many sorts of computing and communication resources, interconnected by abstract *links* and/or *LANs* to form a network topology in which the experiment is carried out. In the original Emulab model, nodes are implemented primarily using general-purpose computers, while interconnections are created using virtual networks implemented by off-the-shelf Ethernet switches. Recent advancements in DETER and Emulab extend this model somewhat.
- Experiments are generally isolated from one another, but make use of support services provided by the testbed infrastructure, such as file systems shared between experiment nodes and an event delivery system that enables loosely coordinated changes to the state of experimental nodes.
- There is a general communication path between experiment nodes and and testbed servers that can be used to remotely access the experiment interactively or programmatically. Depending on experimenter's comfort and familiarity with testbed

services, either standard testbed services or more ad hoc systems may be used to carry out experimental procedures.

The DETER Control Architecture represents and implements a continuing evolution of this basic testbed model.

Given this environment, the basic process of creating an experiment (slice) in the TIED/DETER environment consists of three steps:

1. Acquiring access to individual testbeds consistent with local access control policies.
2. Allocating necessary resources to the experiment using local allocation strategies.
3. Forming the shared experimental connectivity and composing the required experiment services.

Each of these functions is implemented within the control architecture of the DETER/TIED system. Core elements of the DCA are shown in Figure 3 below.

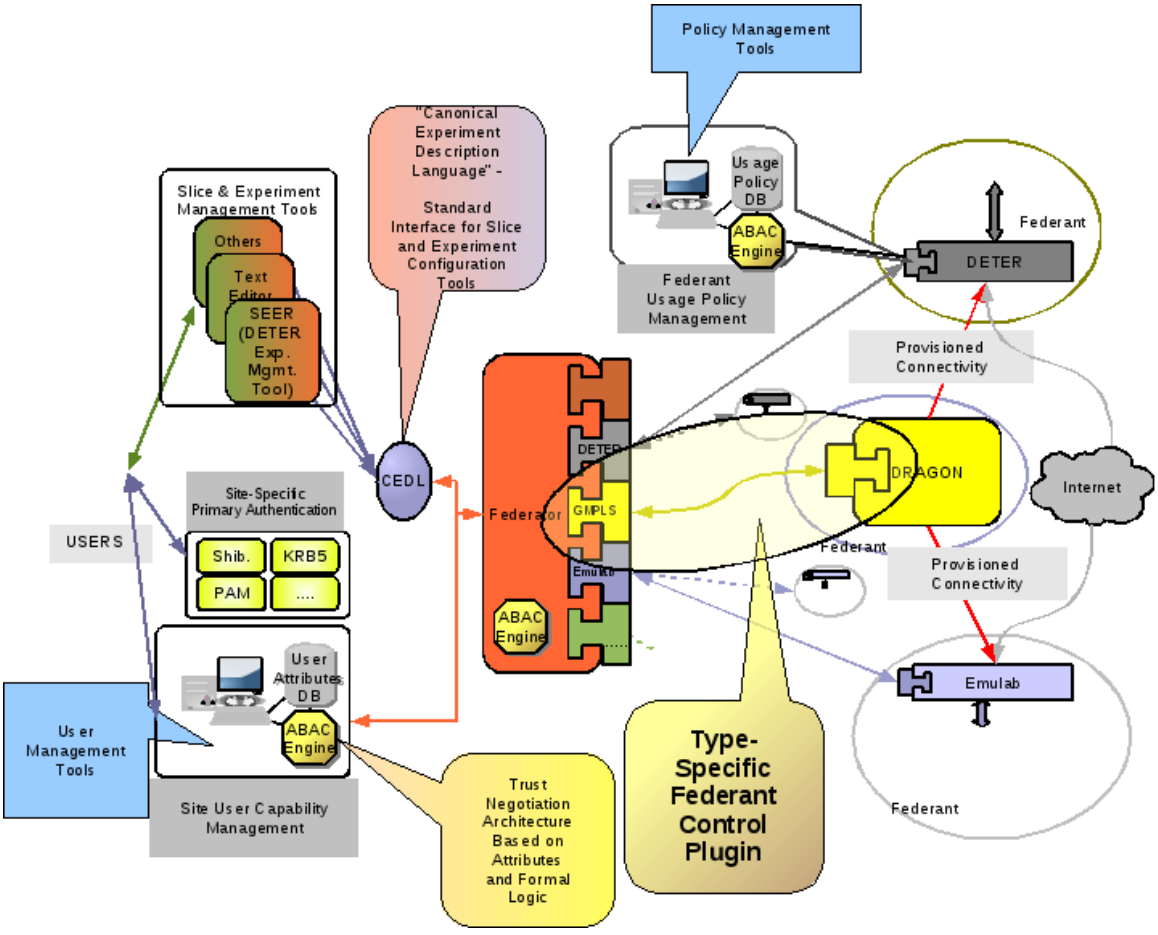

Figure 3: The DETER Control Architecture

Central to the DCA architecture is the *federator*. The federator and its control language, CEDL, serve as a "narrow waist" within the design, providing a unifying functional layer between, on the one hand, a broad range of specialized tools for user interaction and experiment configuration, and on the other, the interconnected resources of multiple physical facilities with different resources and capabilities. The federator acts as the interface between users who are creating and controlling an experiment (slice) and the various federants who have supplied the resources that constitute the slice. It presents users with a single interface for control, but translates the creation of sub-experiments/slivers into the configuration system of the local resource owners.

2.1 The Role of Plug-ins in the DCA

The previous section outlined the core elements of the DETER Control Architecture and described the central role of the federator in this architecture. In implementation terms, the federator is broken up into two parts, the *experiment controller*, which manages the slice or experiment as a whole, and an *access controller* for each testbed or facility that contributes resources. The access controller is responsible for translating the access control decisions from the global domain into local configurations, for using local resource allocation systems to bind resources to the experiment, and for configuring those resources to create the topology and relevant services for the experiment.

Because the function of a access controller is specific to the class of testbed or facility it is controlling, the access controller is implemented as a *plug-in*, with different plug-ins used to interface with different classes of facility. The plug-shaped icons in the figure show the location of these plug-ins within the DCA design. The figure shows that each plug-in supports two interfaces, a standard one to the federator and a testbed-specific one to the testbed itself.[5]

Each plug-in may be implemented and deployed in several configurations, depending on the operational and administrative requirements of the testbed being federated. The experiment controller and access controller may run on the same machine, with the access controller proxying requests back to the testbed it manages; the access controller may be co-located with the testbed and accessed remotely by the experiment controller; or the plug-in may run on a third machine unrelated to either the testbed it controls or the location of the experiment controller. One can think of these layouts as placing more or less functionality in each of the plug-ins in Figure 3.

Though each experiment/slice is controlled by one experiment controller, experiment controllers are fairly lightweight entities. They are responsible for splitting the experiment up between access controllers and managing the credentials of the researchers who are creating the experiments. None of these responsibilities require that the experiment controller run on testbed resources; experiment controllers that run on desktops and communicate with access controllers running on testbed nodes is a likely configuration.

[5]In the figure it appears that plug-in code must be loaded in the federator code base itself, but this is a conceptual diagram rather than an implementation block diagram. What is important is the standard interface between federator and plug-in.

3 The Slice-Based Federation Architecture and GENIAPI

The SFA document traces its history to the earliest days of the GENI development, and the architecture it describes reflects many of the fundamental ideas behind GENI. Its original goal was to define a minimal interface that conformant GENI implementations would export, and it has become a touchstone for the architecture in general.

The GPO-managed prototyping and implementation phase encouraged multiple implementations of the architecture defined in the forerunners of the current SFA document. These implementations were called *control frameworks*. Confronted with the loose definitions of core resource allocation and management functions, data structures, and authorization framework, each control framework implementor made a set of design decisions and created a different system within the overall confines of the architecture. The most successful of these implementations, ProtoGENI and PlanetLab, extended their existing interfaces to include the SFA interfaces. Of course, the underlying resource models of these and other control frameworks strongly influenced how they interpreted and extended the SFA. While the implementations shared a spirit, they differ in significant details.

The GENIAPI project is an attempt to reunify the various control frameworks to the point where they can interoperate. The initial release of this effort is a specification and implementation of the aggregate manager API[14]. That implementation is not a strict implementation of the SFA aggregate manager interface (called the *slice interface* in the SFA document), and is, in many ways, an improvement. We briefly review the current SFA document and the GENIAPI interface.

3.1 The SFA Document

The SFA document defines the key entities, abstractions, and data flow for resource allocation in GENI. It continues to be regularly revised, and the current version includes notes from the authors[5].

The SFA lays out the basic players in the GENI ecosystem and then describes the key abstractions of *slice* and *sliver* that underly much of GENI discourse. The SFA describes a sliver as a collection of co-managed resources and slice as a collection of slivers and users bound to the collection. It describes the life cycle of a slice, all indicative of the slice's central role in experiment creation in GENI.

Components and *aggregates* are also defined as abstractions of co-managed resources that can be multiplexed (*sliced*). The component/aggregate distinction is somewhat slippery, and is based on whether one or more user-visible resources are managed by it. We generally use “aggregate” to refer to this abstraction.

Naming and identification of actors in GENI is discussed in its own section of the SFA document. A fairly intricate system of binding a principal to a public key, a universally unique identifier[15], and a lifetime is put forward. These identifiers are called GIDs. The authors note that none of the implementations have adopted this framework, and the section will be revised,

though we will continue to use GID to indicate a principal identifier. The lack of agreement on naming and identity remains an interoperation problem.

The *Rspec*, *ticket*, and *credential* are laid out as fundamental data types. The Rspec is a resource specification used to request slivers and slices; a ticket is a signed Rspec bound to a GID and a unique identifier used to denote a reservation from an aggregate. Though it is recognized as a fundamental data structure, the Rspec's formats and requirements are not specified here.

Credentials are described as the binding of a particular privilege to a GID, and a set of privileges are defined. Section 8 of the SFA refines this definition, and we discuss the refined definition below.

Following this interfaces are defined, including the aggregate interface also defined and implemented by the GENIAPI. The interface describes the binding of slivers to slices, though the interfaces do not include an explicit parameter for the slice being bound. The binding is described in the functional descriptions. Aggregates are aware of slices and carry out the process of binding resources to them, which corresponds more closely to our recursive model.

The last two sections of the SFA are somewhat different in tone from those described above. Section 7 titled “Authorization and Access Control,” is fairly heavily annotated with disagreements. The authors agree that there are authorities responsible for authorizing access to slice interfaces and separate authorities controlling resource allocation, but there seems to be less agreement on how these authorizations are expressed. Discussions of group rights and individual attributes are included.

The final section, “SFA Authorization Using Registered Capabilities” seems to be an example of realizing the architecture, but there is no such explicit statement. The section introduces a new architectural element – the *registry* – and expands the definition of credentials. This separation of slice manipulation from earlier allocation mechanisms is closer to our two-level model.

A registry contains information bound to GIDs, including human-readable-names (HRNs). The HRN defines a set of entities responsible for validating the identity of the principal having that GID. This separation of validation information from the identity is somewhat confusing, and at odds with the earlier interface definitions. Other data is maintained in the registry, including real-world information about principals and slice information. Interfaces and privileges for accessing the registry are defined.

The description of a credential is expanded upon, defining the credential as a binding between principal GID, object GID (a sliver, slice, or registry), the privileges authorized and their lifetime and an expression of how those rights might be delegated.

The final section's indications of a two-level model conflict with the earlier sections' implications that the aggregate manager binds slivers to slices as in a recursive model. This is one of the indications that the community has not formed a consensus around the overall role of the aggregate interface.

3.2 The GENIAPI Definition

The GENIAPI specification of the aggregate manager interface is an improvement on the SFA document in some respects. The various slice and sliver operations include explicit naming of the object to be manipulated rather than being part of the credential. At points in the interface where data structures are evolving or extensible by nature the GENIAPI adopted appropriate self-describing data structures to support evolution and extension.

The specification is a subset of both the SFA document and the various existing control framework implementations. For example, none of the calls for ticket manipulation exist, which may make writing resource brokers difficult.

This specification does not directly specify either Rspec or credential formats, although some commonality is essential for long term interoperability. While the code picks an identity format based on X.509 certificates, the documentation does not specify that format.

Finally, key operations like creating a slice or assigning credentials remain unspecified. This may be because such definitions are missing from the SFA, because the implementations have adopted different models. These missing interfaces may also reflect confusion between the two usage models we have posited.

4 TIED Plug-ins and the GENIAPI

A TIED plug-in needs to:

1. Map TIED user's credentials into control framework credentials
2. Map requests into local request syntax and semantics
3. Allocate local resources
4. Integrate allocated resources into the shared environment

The standardized parts of the GENIAPI only simplify the third of those four steps, and, in fact, the full SFA is only moderately more useful. Neither defines formats or semantics for identity, credentials, or the Rspec in sufficient detail. As a result, a TIED plug-in using the GENIAPI would still need to be specialized for each control framework.

Considering even the two frameworks that currently most closely implement the GENIAPI, PlanetLab and ProtoGENI, each has their own assignment of identity and mechanism for getting an identity. PlanetLab's HRN-based identity is usable across most PlanetLab as is ProtoGENI's, but a plug-in will use one or the other, and that registration and mapping will be control framework specific.

Slice creation on the two is also materially different. PlanetLab requires a fair amount of information to be installed in a slice record in order to create a slice, while ProtoGENI requires little more than a suggested name.

Similarly a TIED plug-in will need to know if it must make its resource request in a ProtoGENI format or a PlanetLab format. Each is slightly different and each is transparently passed through the GENIAPI. While the same code can make the call to allocate a sliver, the formatting of the request is different.

Finally, initial access to the allocated resources is managed differently on the two control frameworks. Space for some of that configuration is in parameters to the GENIAPI CreateSliver call (user keys), and some in the Rspec (software on allocated nodes).

For a TIED plug-in writer, the issue is not whether a plug-in can use the GENIAPI as defined, but what benefit one might get from using it. Right now, the difference between using the GENIAPI interfaces and using the published control framework interfaces is minimal, because the services provided portably by the GENIAPI are not the difficult part of creating a plug-in. We plan to port the existing ProtoGENI plug-in to use the GENIAPI interface; but we do not expect this to simplify any future PlanetLab plug-in appreciably.

5 The GENIAPI as an Interoperability Architecture

While the current GENIAPI is insufficient to act as an interoperability fixed point, it is the right place to try to enforce portability constraints on control frameworks. The control framework implementers and designers have indicated a willingness to identify commonalities by continuing to improve the SFA document. The GPO's push to standardize a set of useful interfaces that is embodied in the GENIAPI is an indication that the ideas that the community converges on can be moved into code quickly.

We need to both decide what aspects of the interface must be standardized and converge on the requirements and descriptions of them. These choices should be minimally constraining while offering sufficient detail to support interoperation. This process will require dedicated technical and political work. This section discusses our recommendations on further areas of standardization, and how these areas might be standardized. The areas discussed are identity, credentials, slices, and the Rspec. How is the subject of each subsection.

Before proceeding with the discussion, two points should be noted.

First, we note that discussion about standardizing some and perhaps all of these areas has previously occurred in the GENI community, dating back to the original planning group. In some cases, explicit decisions were made *not* to attempt reaching global agreement or standardization at that time. In some cases, the motivations and conditions underlying these past decisions may still currently hold, while in others, they may not, and the potential for a common shared approach may now be stronger, and the costs lower.

Second, we observe that small-s standardization is a social process. Successful standardization implies an ongoing effort by the various GENI designers and stakeholders to work out a common solution in sufficient detail to be useful, but also the recognition that the inability to do so may be due to significant and valuable differences of perspective and vision, rather than “failure”. This action is more in the spirit of the collaboration on the SFA document, than on specifications being handed down from the GPO, though they are an important stakeholder. These

recommendations are intended to influence the direction of collaborative effort, not to indicate a need for administrative fiat. We note that interoperable technical alternatives to standardization of each of the elements we outline are possible, and that in fact the essence of federation is the lack of need to agree to uniformally common standards and policies.

5.1 Identity

The notion of an underlying identity for objects and principals is key to the system, and often overloaded with other system semantics. The authors of the SFA and implementers of the control frameworks seem to agree that identity is key but that we do not yet have a good handle on it.

We call out a few very simple properties that an identity should have:

1. The holder of that identity must be able to independently prove they are the valid holder of it.

2. Two parties referring to the holder of an identity must be certain they are referring to the same entity.

3. The holder of an identity must be able to unambiguously attest to statements or requests as being originated by it.

The first property divorces identity from any specific control framework or identity service. Such services may later tie *facts* to an identity – such as a person's name or institution – but those facts are separate from the identity itself. Independent self-certifying identities - without additional semantic overloading - are key to scaling the system.

The second property insures that the identity is meaningful for reasoning about or delegating rights to. Without this, systems within a framework would have difficulty operating, much less systems between frameworks. Notice that this property does not rule out the possibility that one person has multiple identities. A strict one-to-one mapping between people and identities would be extremely difficult to enforce in practice. It is also sometimes useful for the same person or entity to operate in separate guises.

The third property allows requests for service or delegations of authority to be validated when the holder of the identity is disconnected or unavailable.

A public key of sufficient size exhibits these properties, and is a reasonable implementation of self-certifying identity assuming that the holder of the identity protects and keeps the private key[6]. The holder of a key can prove they are the rightful holder of the key by responding to a challenge encrypted with the public key. As long as the private key is protected and the key size large enough, duplicate keys are essentially impossible, and such a key can sign data in various formats.

[6]Compromise of the private key means that permissions and facts about the old identity must be revoked. Any identity system that relies on a secret has similar problems.

Adopting public keys as an identity and choosing a canonical key format and representation would go a long way toward establishing a GENI-wide identity. Keys can be used in a variety of ways in existing protocols, and can be represented in a variety of ways. Saying that a GENI identity is a 4096-bit RSA key does not prevent one control framework from expecting one to prove identity using a self-signed X.509 certificate and another from using a PGP signature format, but does provide a way that users and administrators can independently create unique identities with proper semantics for GENI operations.

5.2 Credentials

In our view, credentials are less about specific *rights* that the holder of an identity possesses than about describing *facts* about the holder than can be used to make authorization decisions. This enables local resource owners or managers to control how their resources are used through policies based on those facts. Of course, if the idea of explicitly granting privileges for specific operations is attractive, such a specific credential falls within our view.

Currently, we believe credential formats are too loosely defined and their semantics too tightly constrained to be useful for cross-framework interoperability.

Credential formats pose great difficulties for standardization in general, because with current technologies the validity of a signed credential is tied to its formatting. In most cases, reformatting a credential one did not issue is equivalent to forging it. This makes cross-framework translation problematic.

Currently each control framework has its own credential format. Some use standard certificate formats while others use locally defined data structures. Credentials are issued by the control framework and have no meaning – in fact may not be interpretable – in other frameworks.

In practice, some choice of credential format or formats needs to be made. We argue that a few formats that are in wide use should be adopted so that GENI can leverage existing authentication and authorization systems. Shibboleth is an example of an authentication system that both identifies users and attests to attributes about them that are used for authorization decisions. X.509 credential formats are also valuable in that they are widely implemented.

We advocate using the X.509 credential format in order to take advantage of the significant software that supports it. These credentials underly many real world applications and the libraries that support them have been well-tested in the security community.

Rather than using the traditional X.509 hierarchical chain of endorsers, we self-sign certificates. Each certificate is therefore the assertion of a credential by a single principal. Using self-signed certificates allows us to leverage the format and libraries that support it while keeping our lightweight identities.

Before we discuss the semantics to encode in the credentials, consider that within GENI, the meaning of credentials is currently somewhat muddled. The SFA document describes the use of credentials as capabilities, but the analogy is not perfect. Traditionally a capability both references an object and confers rights to that object. These rights are independent of the identity of the capability's holder. The credentials described in the SFA reference an object, but

confer the associated rights only to a particular holder. These credentials are the equivalent of an entry in an access control matrix: they prove a specific user has the right to carry out a set of operations on a specific object.

In the SFA document, the authors use credentials to select the sliver on which to operate as well as to prove that the entity requesting the operation has the right to do so. In particular the operations that start, stop, reset and delete slivers all take only a credential, rather than a sliver identifier and credentials[7].

Furthermore, these credentials do not, in practice, encompass all the information used to make allocation decisions. Elements of the requester's identity (or local facts about the user stored in the registry) may give the requester priority or access to restricted resources. Such implicit authorization information makes understanding the system and auditing decisions difficult because an analyst or auditor can never be sure they have considered all facts relevant to a given decision.

At the very least, the GENIAPI must specify a standard mechanism for acquiring credentials. Even if credentials are only interpreted within one framework and are opaque to the interfaces, users need to have a way to get them.

We believe that it is profitable to go farther and to adopt a credential framework that encodes authorization reasoning and provides verifiable, auditable results. Our credential system of choice is the ABAC system[16] developed by Stanford and Trusted Information Systems (now Sparta), and further developed by ISI and Sparta. It defines an attribute-based model for assigning rights among distributed cooperating principals using simple formal logics. The TIED project wiki includes a more detailed discussion of using ABAC in GENI[17].

Whether ABAC is the credential system the community converges on or not, we believe that there are significant benefits in allowing parties outside any control framework to issue meaningful credentials. For example, an NSF principal could issue credentials identifying some GIDs as GENI developers, and control frameworks could grant permissions based on that fact, rather than a simple credential as defined in the SFA document.

This requires adopting a meaning in the credentials themselves, for example an encoding of ABAC rules within a credential, and a set of formats so that third parties can issue the credentials and be understood. Again, the GENIAPI definition offers an opportunity to specify and support a credential format and meaning, if the community agrees to it.

5.3 Slices

Slice creation and manipulation affects interoperability at two places. The ability to create a slice in a uniform way allows a researcher to create experiments through different control frameworks; cleanly defining how much slice semantics interact with sliver allocation allows an aggregate manager to interact with multiple control frameworks.

[7]The GENIAPI includes a name parameter in the subset of those calls it specifies.

To manage the first, a standard interface for slice creation needs to be defined, either in the unified recursive interface or as part of the control framework in the two-level model. Current implementations of slice creation generally involve creating a registry entry, but the requirements vary widely between implementations. A unified mechanism would simplify this kind of interoperation.

In order for an aggregate manager to offer its resources to multiple control frameworks, it must decouple the operation of creating a sliver from the operation of binding the sliver to a slice. Currently an aggregate manager is aware of the slice to which its resources are bound, and incidentally to elements of the slice's implementation. Because the slice abstraction is created by the control framework, this binds aggregates to specific frameworks.

Sliver creation is tied to slices because GENI designers want to be able to guarantee that slices can be controlled as a single unit, primarily for revocation in the case of misbehavior. For this reason, the creation of a sliver and its binding to a slice were made atomic; all allocated resources are bound to a slice. An unintended consequence of this is that aggregate managers are tightly bound to their control framework implementations to effect the atomic operation. We believe that the behavior can be maintained while removing the implementation dependence.

Maintaining this useful invariant requires standardizing the binding process while insulating the aggregate from slice internals. The atomic action becomes a two-phase commit. A control framework requests a sliver, the aggregate conditionally offers a sliver, the framework binds the sliver to a slice using its implementation and informs the aggregate, and the aggregate finalizes the resource reservation. If the binding fails the resources are released. Concretely, this requires splitting `RedeemTicket`() and `CreateSliver`() into two calls.[8]

Insulating aggregate managers from slice implementation details is crucial to allowing aggregates to offer resources to multiple control frameworks.

5.4 Resource Specification

Historically the Rspec has been a challenging data structure. It has been asked to play many roles in the resource allocation process. Even in the current SFA document the Rspec is used both to request resources and to describe the resources that have been granted, though the two operations require somewhat different semantics. A request could sketch a range of acceptable possibilities while a description of resources granted is necessarily concrete.

A more fundamental matter is that there are at least two philosophically different views of the Rspec present within the community. One view sees the Rspec as essentially quite simple, a straightforward data structure, perhaps extensible in format, but describing only the basic properties of a resource. The other view sees the Rspec as having the properties of a rich semantic description language, building a full ontology of resource descriptions and supporting advanced machine reasoning capabilities. Given these views, it is apparent that no single Rspec format is likely to emerge in the near future.

[8] A recursive model interface would add the two-phase equivalent interfaces and access control them.

These views represent two points on a continuum between very semantically rich and very declarative. As the community moves from Rspecs that enumerate the physical resources that make up an experimental environment to Rspecs that describe the environment in sufficient detail to reason about it productively, there are useful stopping points. For example, moving from enumeration of resources to imposing constrained choices shifts the representation toward richness and adds direct short-term benefits. ProtoGENI Rspecs have already become semantically richer by adding virtual connections and generic node representations. We expect this process of refinement to continue.

However, there are concrete interoperability problems today. Although there have been ad hoc demonstrations of the ability to allocate resources from multiple control frameworks[9], there is no standard way to ask for resources from aggregates attached to different control frameworks.

The GENIAPI command line tools include an "omnispec" format that is a wrapper around the Rspecs used by the various control frameworks. The tool depends on translation modules supplied from each control framework to the command line implementation. Command line tools convert the omnispec to a local Rspec. The tool is less documented than one might hope for, but we view it as a useful stop gap.

The existing Rspec formats are fairly close in format and in the semantic continuum. This would be an excellent time to unify the formats, though we recognize the political and workload constraints that make this difficult. Standardization costs time and effort, and after creating and implementing the framework implementations would be functionally close to where they started. New aggregate manager writers would have a significantly easier time, providing eventual system wide benefits.

6 Summary

This document describes our position that the GENIAPI standardization effort is a valuable step if control frameworks are to easily interoperate. While we believe this goal to be laudable and the mechanism appropriate, the standard is not currently well enough developed for a TIED plug-in to realize material benefits from using it.

We believe that the role of the current API in connecting control frameworks to aggregates is somewhat hazy and have tried to articulate the possible alternative high-level frameworks, a two-level system and a recursive system. While we do not take a position on which is superior, the community does need to pick one.

In order for TIED or other cross framework applications to realize a benefit, the standard must reflect an agreement on identity, credentials, and resource requests. In addition, the semantics and interfaces of cross-framework slices need to be specified.

Once agreement on those issues is in place, the GENIAPI standard should act as an effective interoperability tool.

References

[1]DETER, "The DFA Plug-in Architecture," *http://fedd.isi.deterlab.net/trac/wiki/FeddPluginArchitecture*, 2010.

[2]Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad Mac Newbold, Mike Hibler, Chad Barb, Abhijeet Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," *Proceedings of OSDI*, (October 2002).

[3]ProtoGENI, http://www.protogeni.net/trac/protogeni/wiki.

[4]Thomas Lehman, Jerry Sobieski, Bijan Jabbari, "DRAGON: A Framework for Service Provisioning in Heterogeneous Grid Networks," *in IEEE Communications Magazine,* Vol. 44, no. 3, March 2006.

[5]Larry Peterson, Robert Ricci, Aaron Falk, Jeff Chase, *Slice-Based Federation Architecture*, July 2010, *http://groups.geni.net/geni/wiki/SliceFedArch*.

[6]The GENI API, *http://groups.geni.net/geni/wiki/GeniApi*.

[7]Larry Peterson, Soner Sevinc, Scott Baker, Tony Mack, Reid Moran, Faiyaz Ahmed, *PlanetLab Implementation of the Slice-Based Facility Architecture,* Version 0.07, *http://svn.planet-lab.org/attachment/wiki/WikiStart/sfa-impl.pdf*, Sept. 2009.

[8]OpenFlow, *http://www.openflow.org*, 2010.

[9]Guido Appenzeller, "OpenFlow Demos at GEC8," *http://www.openflowswitch.org/wp/2010/07/gec8/*, 2010.

[10]Tom Mitchell, *GENI Aggregate Manager API*, (slides), http://groups.geni.net/geni/attachment/wiki/Gec8ControlFrameworkAgenda/GEC8-Mitchell-AggregateManagerAPI.pdf, 2010.

[11]How to use ProtoGENI, http://www.protogeni.net/trac/protogeni/wiki/Tutorial, 2010.

[12]Users Guide to the SFI, http://svn.planet-lab.org/wiki/SFAGettingStartedGuide, 2010.

[13]Ted Faber, John Wroclawski, *TIED Testbed Control Framework: Plug-in Design Document*, Feb 2010, *http://groups.geni.net/geni/attachment/wiki/TIEDProtoGENIPlugin/TIED_CF_plugin_design_spec_v1.0.pdf* .

[14]*GENI Aggregate Manager API*, *http://groups.geni.net/geni/wiki/GAPI_AM_API*, 2010.

[15]*International Standard "Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components"* (ITU-T Rec. X.667, ISO/IEC 9834-8, 2004.

[16]Ninghui Li, John C. Mitchell, and William H. Winsborough, "Design of a Role-Based Trust Management System," *in Proceedings of the 2002 IEEE Symposium on Security and Privacy*, (May, 2002).

[17]ABAC Authorization Control Model And Discussion, http://groups.geni.net/geni/wiki/TIEDABACModel, 2009.