TIED Testbed Control Framework: Plug-in Design Document

Ted Faber, USC/ISI

John Wroclawski, USC/ISI

Draft Version 1.0

Feb 3, 2010

Table of Contents

1 Introduction

This document performs two functions:

- It specifies the functional requirements and programming interfaces for resource management plug-ins within the TIED/DETER Control Architecture (DCA). Within the DCA, resource management plug-ins provide the interface between the core control framework and individual facilities (federants) of different types, allowing federants to contribute physical resources to a TIED slice / DETER federated experiment. The document proceeds from high level concepts to detailed interface descriptions. This work is evolving, and we will publish updates to this specification as stable points are reached.

- Building on the functional and programming interfaces outlined above, it describes the design of a ProtoGENI[1] plug-in for the TIED system. This plug-in enables the creation of slices that span ProtoGENI and TIED-enabled testbeds. ProtoGENI is of particular interest both because it is useful in its own right and because it implements an interface that is expected to bear significant resemblance to the initial "converged" GENI API. A working ProtoGENI plug-in is a significant stepping stone toward a plug-in for the converged API, as well as providing tests of both the TIED/DCA plug-in architecture and that API in both design and implementation phases.

The remainder of this introduction provides an overview of the TIED/DETER Control Architecture sufficient to understand the role of DCA plug-ins within this architecture. Section 2 presents an overview of the DCA plug-in functional specification, while Section 3 gives the functional specification itself. Section 4 presents a detailed design specification for the DCA ProtoGENI plug-in, currently under development.

1.1 Overview of the DETER/TIED Control Architecture

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:

- An experiment consists 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,[1] while interconnections are created using virtual networks implemented by off-the-shelf Ethernet switches.

- Experiments are generally isolated from one another, but make use of support services provided by the testbed infrastructure, such as file systems

[1]Some support for non-computer nodes and simulation options is also available.

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. Key capabilities of the current architecture include:

Support for federated infrastructure. The DCA is generalized to allow multiple testbeds to contribute nodes and interconnectivity while maintaining the node/interconnectivity abstraction and and the concept of experiment-wide services. Further, each testbed retains substantial control over its own resources and usage policies. When available, a federated experiment may use dynamically provisioned wide-area networking infrastructure to provide service guarantees. When such facilities are unavailable, performance between testbeds, either for in-topology traffic or exported services, is best effort.

Support for heterogeneous and abstract, virtualized experiment elements. In ongoing work, the DCA is being extended to support a) new classes of physically realized experiment elements, such as switches, firewalls, reconfigurable hardware platforms, etc., and b) logical or “virtualized” experiment elements modeled or emulated using a variety of techniques.

This second class of objects are not emulated directly by physical hardware, but are virtualized, under a very broad definition of “virtualization.” In our use of the word, virtualization is the representation of individual experiment elements using whatever technology is appropriate to meet necessary experiment requirements and invariants, including scaling to appropriate size while maintaining required behavior. Thus, the “virtualization” technique for a particular experiment element may range from a full VM implementation to a lightweight sandbox to a simple thread of execution. In each case, however, the logical element is viewed and manipulated similarly by the control architecture.

Support for an expressive, formally verifiable usage policy and access control infrastructure. To facilitate the use of multiple, federated infrastructure facilities operated by different entities, and support a broad community of researchers using these facilities, the DCA supports a rich, expressive, and formally grounded attributed based access control model (ABAC)[3].

Support for experiment services in a heterogeneous, federated environment. The Emulab experiment model includes the concept of testbed-wide services, as well as the basic infrastructure of nodes and links dedicated to an experiment. In the heterogeneous, federated environment implemented by the DCA, support for this concept remains extremely useful but becomes more difficult. As experiment elements become less like general-purpose computers, ideas of exporting shared file systems or user accounts become less universal. What it would

mean to have a shared file system available to a cloud-based real-time simulation of a botnet is hazy at best.

Beyond this basic issue, from the point of view of a federated system, the services supported by different underlying testbeds may be significantly different or incompatible. Yet, we still wish to provide coherent services across these heterogeneous facilities. This suggests a compositional approach to experiment services, where individual testbeds characterize the services that they can export or import and the control architecture builds appropriate environments from them.

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 1 below.

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.

1.2 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.[2]

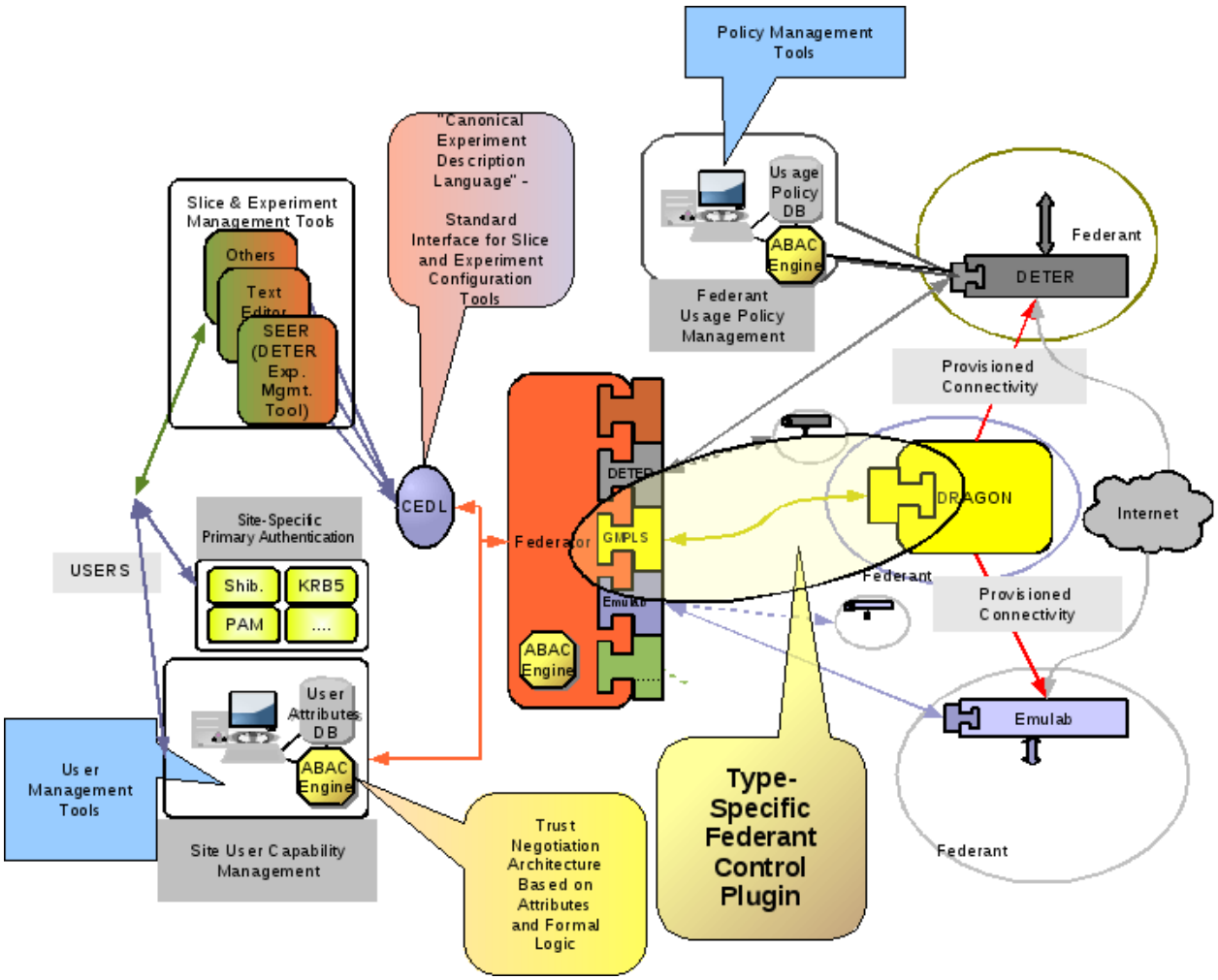

Figure 1: The DETER Control Architecture

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

[2] In the figure it appears that plug-in code must be loaded in the federator codebase 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.

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 plugs in Figure 1.

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.

The next subsection describes the key features of the plug-in interface and later sections describe its current state in detail.

2 Overview of the plug-in specification

The motivating features of the plug-in architecture include a topology and connectivity representation that is extensible and simple to parse, explicit description of experiment services using an extensible format, and incorporation of a flexible, expressive, and formally verifiable attribute based authorization system. Each of these capabilities is discussed briefly below.

2.1.1 Sub-Experiment Representation

The purpose of a sub-experiment representation, or description language, is to describe the resources and interconnection topology of a sub-experiment – the part of an overall experiment to be allocated to a single facility or testbed.

Earlier versions of the DCA used straightforward extensions of Emulab's experiment description language, based on ns-2[4], to describe sub-experiment topologies. This was a natural design decision when federation primarily concerned itself with combining Emulab-like resources. The current specification uses a more extensible model to describe experiments in a declarative language.

The ns2 description language describes experiments as nodes that are tacitly assumed to be configurable general purpose computers, connected by links or LANs. Nodes can have arbitrary disk images loaded on them, but are essentially assumed to have hierarchical file systems and to support the Emulab event system and shared environment. There are exceptions to these rules, but the underlying assumptions color the design.

In addition to the underlying bias toward general purpose computers in the Emulab description language, ns-2 is an extension of tcl[5], which is a Tuning-complete programming language. Emulab includes libraries for generating declarative representations from the ns2 programs users upload, but asking each testbed to translate an arbitrary tcl program into its local resource representation imposes requirements that not all testbeds are able to meet. More importantly, the procedural turing-complete nature of the ns2-based description language makes

certain formal analysis and verification actions on experiment descriptions extremely difficult.

Rather than ask access controllers to run programs to create Emulab-centric representations of sub-experiments, we have specified a declarative, extensible format for expressing sub-experiments called *topdl* (short for topology description language). A complete definition follows later in the document.

Topdl represents general resources that can communicate with one another. A resource might be a general purpose computer, or a testbed or sub-topology. If two resources can communicate directly (logically), they share a substrate. Every resource that is connected to a substrate has an interface to that substrate; a resource may have multiple interfaces to the same substrate. Substrates represent logical connectivity.

There are several types of resource defined, and more can be added or derived from existing definitions. In addition all resources can have simple attribute/value pairs attached to them without defining new types. The attributes allow fast prototyping while the full derivation system allows tested ideas to be integrated easily.

Similarly interfaces and substrates have a few simple properties defined and can be extended using either mechanism. We plan extensions of substrates to support wireless interconnection spaces.

Topdl is defined in XML, allowing systems to take advantage of the many tools available to simply and effectively parse them. We have integrated topdl into our existing system.

2.1.2 Service Model

Experiment support services are treated as first class entities, much as resources. In addition to advertising, allocating, and configuring resources used in the experiment topology, testbeds advertise, allocate and configure services.

While services and resources have significant similarities, there are key differences as well. A service often creates an aspect of the experimental environment that is global with respect to the experiment. Examples might include services for delivering experiment events throughout the experiment, or services that synchronize the clocks of experiment elements across multiple testbeds. These services provide properties that go beyond the hardware donated to the experiment. Cooperation between testbeds is required.

Creating a service that covers more than one testbed's contribution to an experiment requires a composition of the implementation of the services on the federant testbeds. Individual testbeds may fully implement a service locally, in which case extending the service more widely may be a matter of reconfiguring the servers inside the testbed. For example, extending a shared file system or time synchronization can fall into this category. Some testbeds will implement a subset of the function that can be stitched, modified or expanded. TIED's prototype implementation of a distributed experiment-wide event system uses this strategy.

Finally, some services will be completely nonexistent in the native implementation of most testbeds. DETER's SEER experiment control framework is not implemented by most testbeds, but we have been able to import it successfully. There is also the problem that some services are not supported semantically by some resources; a shared filesystem may not make sense to a fiber multiplexer.

The process of creating the specified services from the capabilities inherent to the testbed is the responsibility of the access controller for that testbed. It is responsible for configuring partial implementations to cooperate with other partial implementations or to configure clients to talk to outside servers. More complex configuration is also possible.

The current design for service support is the first step into this complex area. It allows testbeds to indicate the services they support as exporters or importers (where completely decentralized services are entirely imported) through their access controllers and provides a way for the experiment controllers to request and configure those services. In the same way that *topdl* resources can be extended, service representations can be extended. Support for composition of resources is planned for the future.

2.1.3 Support for Authorization

The DCA incorporates an expressive, flexible, and formally verifiable access control system for testbed resources known as ABAC[3]. ABAC is an attribute-based system for making robust, provable authorization decisions in a distributed environment. In the DCA, experiment controllers use ABAC to vet researcher requests, and access controllers use ABAC to authorize allocation of resources and distribution of services.

Because most testbeds and resource providers already implement some form of access control, is expected that most access controllers will wish to build on these preexisting access control mechanisms. Such access controllers will be integrated with their local testbed's authorization system, and effectively be converting from the global ABAC based decisions into local decisions and vice versa.

ABAC is largely predicated on delegation. Widely recognized entities, like the National Science Foundation or a GENI operations office, can provably vet institutions or individuals, assigning attributes to them that may confer standing or authority. Individual testbeds may then make decisions based on these assigned attributes. As a slice or experiment is created, the researcher creating it will delegate some of his or her authority to the slice and allocation decisions will be based on that delegation.

These delegations and attributes are all represented in signed credentials. With an appropriate set of signed credentials, actors (like the controllers) can make provable decisions. There are provisions to gather missing credentials if they exist, but the proofs are most efficient when the credentials required are in the actor's possession.

The implementation of the ABAC control software is ongoing, but the plug-in dataflow is organized to provide the relevant delegation points and pass the credentials to the points making decisions.

3 Functional Specification of TIED/DCA plug-ins

This section of the document lays out the basic plug-in design, with pointers to more detailed specifications when applicable. Section 3.1 describes the overall workflow and semantics of operations. Section 3.2 describes the new sub-experiment representation and Section 3.3 discusses the emerging services description model. Section 4 describes the design of the ProtoGENI plug-in.

3.1 Message Flow

This section describes the transactions between the experiment controller and access controller throughout an experiment lifetime. The first section presents the general overview of the phases of experiment creation and subsequent sections describe the specific information exchanged in each operation.

These transactions are all defined as WSDL interactions between two web services, because that allows significant flexibility in service placement and implementation language. All the transactions are basically remote procedure calls, again for simplicity and flexibility. We have implemented several different controller layouts already as well as demonstrating communication across implementations of sub-components written in C++, Java, and python.

This section describes the interactions at a fairly high level, but the full protocols are given in commented WSDL. That specification can be found at: http://fedd.isi.deterlab.net/trac/browser/wsdl/trunk/ .

3.1.1 Message Exchanges

The first step in creating an experiment is to create the slice that will have resources attached to it, shown in Figure 2. In ABAC terms, this is a principal to which researchers can delegate credentials used to make access control decisions. This is an operation between the user and experiment controller. The controller creates an new identity under which it will request resources and informs the user of that identity.

Figure 2: CreateSlice Exchange

The user then delegates authority to that slice using ABAC. The particular mechanisms used are orthogonal to the rest of the experiment creation function.

The DFA does provide a mechanism to pass the credentials through from researchers to controllers.

The plug-in has nothing to do in this phase but we include it to set the stage for other exchanges.

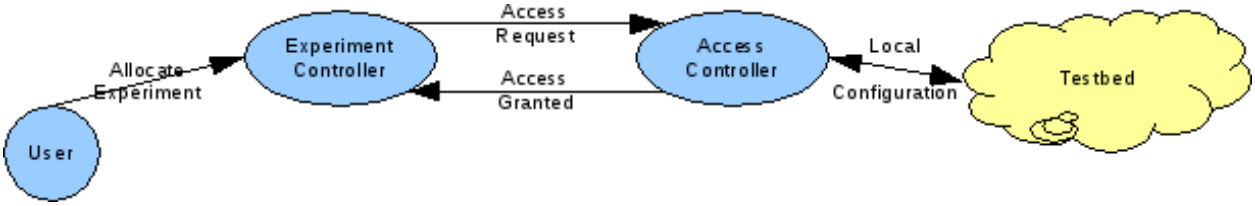

Figure 3: AccessRequest Exchange

When the user begins to attach resources to the slice – to populate the experiment – several exchanges take place, the first of which is shown in Figure 3. There is the initial request from the user, which contains the experiment layout and services requested, which the experiment controller processes. This request includes credentials that the user has delegated to the slice.

The experiment layout can be in the legacy ns2 format, or the topdl format described below. The topdl layout allows user tools to annotate experiment elements with attributes that the experiment controller will pass transparently to the testbeds. This provides a path for tools to include testbed-specific requests in their federated experiments. Tools that know a specific testbed's access controller understands an annotation can include it and get the service implied. This provides a fast prototyping path that does not require every new attribute to be understood by the experiment controller.

Currently most of the splitting and choices of where to embed which components are included in this request, but more sophisticated experiment controllers will make more of those decisions in the future. For each testbed that the experiment controller will get resources or services from, it contacts that testbed's access controller with and Access Request. That request contains a summary of the resources and services of interest to the slice and credentials relevant to the request. This request is made as the slice created in the initial exchange, so the credentials include any that the user has delegated to the slice.

The access controller uses the authorization system to decide if the request is acceptable, and if so configures the testbed to support later requests from this slice. This configuration may be significant, depending on the nature of the local testbed. For example our prototype is capable of creating a new Emulab project on the local testbed, which can require a few minutes.

These access requests proceed in parallel, reducing the overall time to meet the access requirements.

Of course an access request may fail. The current simple prototype treats this as a failure to configure the slice, but more sophisticated future experiment controllers will retry on equivalent testbeds, if they exist.

Once the Access Requests have all completed, the experiment controller knows that it has appropriate authority on the various testbeds to make the actual allocations.

At this point the experiment controller informs the user that the experiment is being populated. Throughout the rest of the allocation the user software can poll the experiment controller for the status of the experiment, and get partial allocation logs or error messages. Because configuring testbeds can be a lengthy process – tens of minutes on Emulabs that have to reallocate an experiment segment several times due to partial failures – we prefer to do the allocation asynchronously.

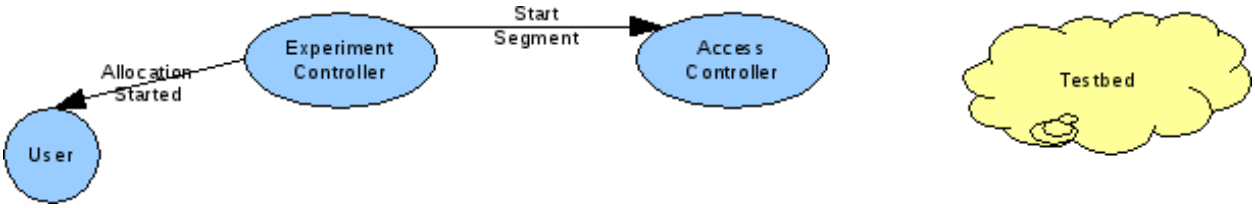

Figure 4: StartSegment Exchange

Now the experiment controller starts the individual segments of the experiment, shown in Figure 4. The request to start the experiment includes the topdl description of the sub-topology as well as configuration information in the topdl attributes. Service configuration information is also included.

In particular, the sub-experiment topologies can contain additional elements not specified by the experimenter that are used to establish in-experiment connectivity between sub-experiments and that forward or synthesize services used by the experiment. The configuration of these elements is accomplished using the features of topdl described below.

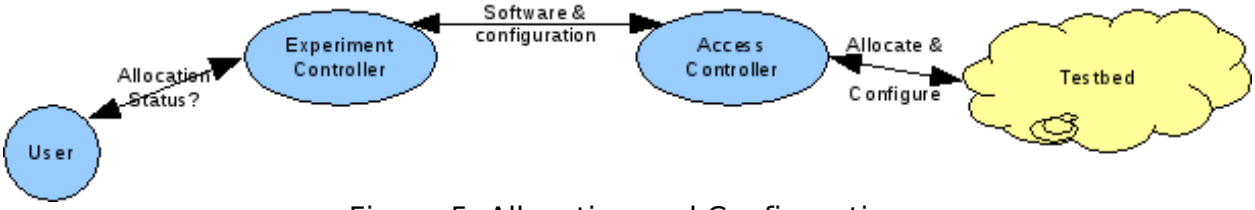

Figure 5: Allocation and Configuration

Starting a segment consists of allocation and configuration of the testbed resources using the local credentials established at access time, as in Figure 5. This may include downloading software distributions and configuration files from the experiment controller. The experiment controller makes this software available over secure transfer protocols with authorization to access the files given by the authorization control system. This phase allows tools and services to be established on testbeds that do not locally maintain the software, while the access control respects the privacy of confidential implementations.

During this process the user software is likely polling the experiment controller, which provides information about the starting segments. As sub-experiments are fully configured and activated, the allocation logs are returned (see below). When a

user queries the state of the experiment allocation, the most recent log information is returned to them as well.

To the extent possible, this sub-experiment starting is carried out in parallel. However some scheduling constraints may appear. For example, transit networks that allocate particular tags dynamically – VLAN identifiers – may need to report that allocation before sub-experiments connected by them are started.

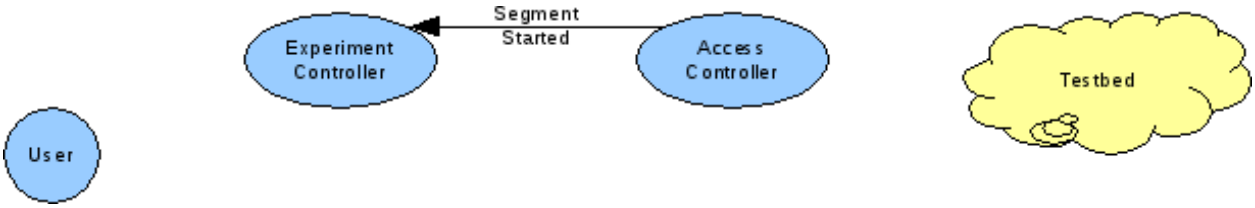

Figure 6: StartSegment completed

As each sub-experiment is activated, the access controller informs the experiment controller that it has been incorporated into the slice, as in Figure 6. There is no particular message sent to the user on completion; but each poll includes the state of the experiment as a whole. When all segments have started, the experiment state becomes active; if a sub-experiment fails to start, the experiment controller cleans up to the extent possible and reports the status as failed. The user code can gather all the logs of the failed experiment from a status poll.

Though we have not diagrammed it, the process of destroying a slice is the reverse of creating it, with many of the configuration steps removed. Each segment is removed from the federated testbed in parallel, and then the access granted to each testbed is removed. Exactly what removing access means to each plug-in depends on the testbed type and administrative choices. Some testbeds will remove projects or other data structures, some will simply note that the account is not currently in use.

3.1.2 Messages and Operation

This section discusses the message fields in the exchanges above in more detail. The canonical definition of the data structures used is available in the XSD stored at http://fedd.isi.deterlab.net/svn/wsdl/current/fedd_types.wsdl . In the descriptions below we give the name of the complexType that contains the message schema. This section outlines the schematics of those messages.

The interactions above mostly come in request response pairs, where the response indicates a success. When a request fails, a fault is sent rather than a response message.

Faults

A fault has three fields:

Name	**Function**

<table>
<tr>
<td>Code</td>
<td>An integer indicating the general type of fault. Valid values are:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>1</td><td>Access denied. Pretty self explanatory</td></tr>
<tr><td>2</td><td>Proxy error. Older versions of the software proxied more requests, and this error indicated that an attempt to forward a request to a proxy had failed. This error is rare in current code.</td></tr>
<tr><td>3</td><td>Badly formed request. In the unlikely event that the SOAP encoding allowed a mis-formed message to be sent, this error is set. Generally it means a required field is absent</td></tr>
<tr><td>4</td><td>Server configuration error. The server is unable to process the request due to misconfiguration.</td></tr>
<tr><td>5</td><td>Internal error. Something very unexpected has happened at the server, such that no other error code is applicable</td></tr>
</table>
</td>
</tr>
</table>

	<table><tr><td>6</td><td>Partial instantiation. A request has been made on an experiment that has not been fully allocated and configured yet.</td></tr><tr><td>7</td><td>Federant error. One of the federants has failed to complete its part of the operation. This error commonly indicates a resource shortage when allocating and configuring an experiment.</td></tr><tr><td>8</td><td>Connect error. Unable to contact a controller. This may be a direct error or an indirect one. For example, if an experiment controller cannot reach a federant, this error will be returned to the user. It will also be returned if the experiment controller cannot be reached.</td></tr></table>
Errstr	A string containing the error class. A simple second representation of the code that avoids all applications needing to replicate the table. Optional
Desc	More detailed natural language

	explanation of the error.

Access Request and Access Response

The AccessRequest contains the following fields:

Name	**Function**
Credential	The credentials attached to this slice. (ABAC credentials are signed, so these can be trusted). Multiple credentials may appear.
Resources	A summary of the resources to be requested. This field is optional and its formatting is in flux as it is generalized away from representing Emulab nodes.
Services	A summary of the services to be imported and exported to this testbed. Format in flux.
AllocID	A unique identifier to reference the access being requested. Generally a new fedid (that is a new public key) is used.
When	A date/time at which to request access. Optional and ignored by current prototypes.
Until	Duration of access if known. Optional and ignored by current prototypes.
ExportProject	Deprecated optional field indicating that this Emulab's project is to be exported to the other federants.

In practice the experiment controller picks the allocationID and sends a request with appropriate service and resource fields included. If access is granted the plug-in creates any underlying structures needed for access – e.g., Emulab projects or users

– and returns an AccessResponse message. If access is denied or the configuration fails a fault is thrown explaining the issue.

The AccessResponse contains the following fields:

Name	**Function**
AllocID	The allocation ID of this access context
When	Start of the validity of the access. Optional and unused by current prototypes
Until	End of the validity of the access. Optional and unused by current prototypes
Emulab	Deprecated and optional mechanism for passing configuration information to the controller.

This message confirms the access information.

StartSegmentRequest and StartSegmentResponse

The StartSegmentRequest contains information for establishing a sub-experiment under the given access allocation. Much of the information is contained in the experiment description field, which is basically a topdl description of the experiment topology. Section 3.2 describes topdl in detail.

Name	**Function**
AllocID	The access allocation to use for this sub-experiment; this slice's right to access this allocation is checked.
SegmentDescription	The topdl description of this sub-experiment. A legacy extended ns2 format is legal to send in this field as well, but not all access controllers will be able to interpret it. The extended ns2 format is deprecated.
ServiceDescription	Description of the services exported by this sub-experiment. The format of this field is in flux.

Master	A boolean indicating that this testbed should export Emulab services to other Emulabs. This field is deprecated in favor of explicit service descriptions and should be set to false.
FedAttr	A set of string-based attribute value pairs that allow segment wide parameters to be set. Generally unused as yet.

The response to this includes the locally formatted log of instantiation and a few other fields.

Name	**Function**
AllocID	The allocation ID again for confirmation
AllocationLog	The locally formatted log of allocation decisions. Usually reported to the user in status messages
FedAttr	Optional string-based attribute value pairs indicating sub-experiment wide parameters. The main use for this field has been to report the VLAN tag selected by a DRAGON transit network.

Software Downloads

As part of segment configuration, access controllers may need to acquire software or configuration files from the experiment controller coordinating the experiment. The location of that software is specified by a URI in the segment description. That URI points back to a repository run by the experiment controller that uses a TLS-secured connection to confirm the access controller's fedid and only allow access controllers involved in the experiment to retrieve it.

TerminateSegmentRequest and TerminateSegmentResponse

Commands to terminate a segment are analogous to those used to start a segment, but simpler because no information about the segment's composition is required. The TerminateSegmentRequest includes:

Name	**Function**
AllocID	The segment to terminate, by its

	allocation identifier.
Force	A boolean. Tells the access controller to do all it can to remove the segment, even if the segment is in an indeterminate or erroneous state.

The TerminateSegmentResponse is equally simple:

Name	**Function**
AllocID	The allocation identifier of the terminated segment.

ReleaseRequest and ReleaseResponse

These messages coordinate the removal of access to the testbed. In the simplest case, these remove only some small overhead keeping track of the allocation identifier, but if more local resources – e.g., Emulab projects or user accounts – have been created, those are removed as well. The messages themselves are straightforward.

The ReleaseRequest contains:

Name	**Function**
AllocID	The access to remove

RequestResponse contains:

Name	**Function**
AllocID	The access removed

3.2 Topology Description

Topdl is the encoding of a simple but expressive system for describing experiment topologies in terms of elements and their ability to directly communicate with each other. The format is used to communicate much of the information needed in constructing sub-experiments – or slivers – in a way that is testbed-independent and extensible. Plug-ins need to parse and interpret topdl to make TIED sub-experiments.

The elements describe parts of an experiment at different degrees of detail appropriate to different phases of operation or provisions of different services. For example, a testbed that is emulating an enterprise network needs to know the types of computers and their interconnections in some detail. A testbed providing a

wide-area transit connection between testbeds may not need to know anything about the internal configuration or even capacities of the testbeds.

Specific resource types do have parameters appropriate to them, but the general goal of the format is to allow annotations and other extensions as simply as possible. We expect this extensibility to facilitate fast prototyping of access to new testbed features or new experiment properties.

Topdl is an initial attempt to meet these needs. It is currently replacing the ns2 experiment representation. As a result of being slotted into that position the element descriptions that are best fleshed out are computers and networks. Some other elements are specified and more are being created.

The following sections break out the specific parts of the format in sufficient detail to understand the basics. The full definition of topdl and its encoding in XSD is available from http://fedd.isi.deterlab.net/trac/browser/wsdl/trunk/topdl.xsd . Code that implements a set of objects that follow the hierarchy is available from http://fedd.isi.deterlab.net/trac/browser/fedd/trunk/federation/topdl.py .

3.2.1 Elements

Elements are the active, configurable parts of the experiment. Examples include computers, routers, and multiplexers. At different levels of detail they can include segments (sub-experiments) or testbeds.

Initial Elements

The defined elements currently include others (a base class), computers, testbeds, and segments. The *other* element includes:

Name	**Function**
Interfaces	The list of interfaces to various substrates. These are described below.
Attribute	A set of string formatted attribute value pairs.

As *other* is a base class, any element in topdl can access a substrate and may have extension attributes attached to it.

A *computer* element has these properties and:

Name	**Function**
Name	One or more strings that allows reference to this computer. It may be inserted into an experiment name table or database as well for experiment

	control.
CPU	Descriptions of processors on the computer. Can include co-processors or other “thinking” elements.
OS	Valid operating system choices for this computer. This field includes both provisions for an operating system version and a standard software distribution.
Software	Additional software installations (outside the standard distributions included in the OS field). Software is given by a URI. Software formats that do not include a destination directory in the format have a spot for the location here.
Storage	Memory and persistent storage requirements.

Valid values of these fields are left somewhat free-form so as to allow access controllers some leeway in allocating hosts, as well as the ability to quickly incorporate new technologies.

Each of these subfields has a more complex type, accessible from the XSD description. They are mostly straightforward, so rather than rehashing them here, we direct the reader to http://fedd.isi.deterlab.net/trac/broswer/wsdl/trunk/topdl.xsd .

We note that each of these subfields can have name/value attributes attached to them. This provides an escape to local testbed or plug-in semantics. For example, an attribute attached to an OS field might tell an Emulab plug-in to use a specific standard disk image rather than leaving the translation from OS to disk image up to the access controller.

A *segment* element encapsulates a sub-experiment that has either been instantiated or is planned. This is used when passing topology descriptions to testbeds that provide network transit services. It includes the following fields in addition to those provided by *other* elements.

Name	**Function**
ID	The federated identifier of the segment/sub-experiment.

Type	A string identifying the basic type of the segment. Primarily used to differentiate transit sub-experiments from sub-experiments that contain elements.
URI	The URI on which the segment's access controller can be reached.

A *testbed* element is primarily intended for use in advertisements or service descriptions. A testbed that provides transit service between a set of testbeds may represent itself as a substrate that connects testbed elements. It has the same fields as a segment element, but without the ID field.

Expanding Elements

Adding new elements is a matter of defining new encodings and attribute meanings and publicizing them. The ability to attach attributes at many places on elements is intended to enable simple extensions that eventually become formalized as new element types. We expect several new element types to appear out of our ProtoGENI prototyping.

3.2.2 Substrates

A *substrate* is a region of logical connectivity of elements. Elements that share access to a given substrate can send messages to each other directly. This is intentionally vague, as one researcher's idea of logical connectivity might hide a set of lower level systems that are providing that connectivity, e.g., the Internet. Another researcher might consider elements to share a substrate only if physically connected.

Though substrates include attributes that describe bounds on their performance, they are generally optional and, when present, constitute upper bounds. This is intended to make substrates suitable for indicating the presence of current simple interconnection technology, like physcial fiber or copper links, while providing plug-in creators and researchers places to hang other attributes that describe more unusual interconnections.

Any description of this nature is a compromise, and topdl substrates are not ideal for modeling all interconnectivity. However, we believe that a broad number of useful technologies can be described currently and more will be added.

A substrate includes:

Name	**Function**
Name	Used by interface objects (below) to bind an element to this substrate.

Capacity	A description of the maximum information carrying capacity of this substrate. Optional
Latency	A description of the maximum latency properties of this substrate. Optional
Attribute	A list of name/value pairs that define other properties of this substrate.

The capacity and latency properties can describe the capacity and latency either as simple maxima or using simple statistical limits – e.g. a peak and average. As the specification evolves we expect to add other ways to define these attributes. The syntax of these fields is available from http://fedd.isi.deterlab.net/svn/wsdl/trunk/topdl.xsd as well.

Substrates are less specific than elements, but similar paths will be used to expand them If need be.

3.2.3 Interfaces

Interfaces tie elements to substrates, in effect defining which elements can directly communicate. In addition to indicating that an attachment exists, an interface can constrain the same performance properties that a substrate defines. For example, a substrate representing line of sight interconnectivity on a particular radio frequency might be upper bounded by the theoretical upper bounds on that frequency while individual interfaces on elements would reflect the limits of current technology, or even location.

An interface is attached to one and only one element, but may be attached to multiple substrates. This may be an appropriate way to model a system that can transmit on multiple wavelengths on the same fiber, for example. The additional performance bounds are applied per-interface.

An element may have multiple interfaces to the same substrate. What this means, and indeed, whether it makes sense at all, depends on what the substrate is modeling.

Specifically an interface includes:

Name	**Function**
Substrate	A list of substrate names to which this interface is attached. At least one name must appear.

Capacity	A bounding capacity in the same format as a substrate capacity. Optional.
Latency	A bounding latency in the same format as a substrate latency. Optional.
Attributes	The list of name/value pairs.

An interface is tied to an element by appearing in that element's interface property. New interface properties can be created by attaching attributes that specific plug-ins respect.

3.2.4 A Topdl Example

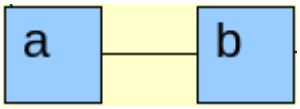

Figure 7: Simple Topology for topdl Example

Here is a simple example of two computers connected by a link in topdl (Figure 7):

```
<experiment>
 <substrates>
  <name>link0</name>
  <capacity>
   <rate>100000</rate>
   <kind>max</kind>
  </capacity>
 </substrates>
 <elements>
  <computer>
   <name>a</name>
   <os>
    <attribute>
     <attribute>osid</attribute>
     <value>FC6-STD</value>
    </attribute>
   </os>
```

```
   <interface>
    <substrate>link0</substrate>
    <capacity>
     <rate>100000</rate>
     <kind>max</kind>
    </capacity>
   </interface>
  </computer>
 </elements>
 <elements>
  <computer>
   <name>b</name>
   <os>
    <attribute>
     <attribute>osid</attribute>
     <value>FC6-STD</value>
    </attribute>
   </os>
   <interface>
    <substrate>link0</substrate>
    <capacity>
     <rate>100000</rate>
     <kind>max</kind>
    </capacity>
   </interface>
  </computer>
 </elements>
</experiment>
```

A more complex example is available from http://fedd.isi.deterlab.net/trac/wiki/FeddPluginArchitecture .

3.3 Services

Of the various plug-in aspects, the services architecture is the most in flux. Initial versions of the rest of the interfaces described in this document have been tested and used to create federated experiments, but the service factoring interface is, as yet, a paper design.

Currently we have a simple model of services in that a given access controller knows how to import, export or integrate a service with other instances of that

service to create testbed-wide services. Service contact points are given as URIs and included at the various phases of experiment creation, described in Section 3.1.1. An access controller that cannot implement a service request or that is unable to import, export, or integrate a service indicates failure to create the sub-experiment.

This starting point buries a fair amount of semantics in the names of the service, and we expect to make those semantics more explicit as this interface evolves. In particular we are interested in making services composable.

4 Design Specification for the ProtoGENI Plug-in

This section describes the design of a plug-in to support ProtoGENI[1] as a TIED federant. The ProtoGENI plug-in implements the functions of a TIED plug, and hence allows the TIED control framework to directly access and integrate ProtoGENI-controlled resources into a federated experiment, described in Section 3.

The ProtoGENI plug-in is the third to be developed for use with the DCA. Existing plug-ins provide support for testbeds that export the Emulab interface, and for DRAGON[6] networks that export an OSCARS interface[7]. These plug-ins provide features (code) that can be re-utilized in the ProtoGENI plug-in. However, the ProtoGENI plug-in is significantly more complex than either of the two existing plug-ins, due to the requirement to map more complex and more heterogeneous semantic operations to the DCA model.

The major design issues for the plug-in include:

- How to map between global TIED authorization attributes and local ProtoGENI identities and credentials.
- How to convert TIED topdl descriptions into ProtoGENI RSPECs and allocate ProtoGENI resources.
- How to interconnect with other TIED sub-experiments using ProtoGENI testbed resources.
- How to provide TIED experiment services using ProtoGENI services and resources cooperatively with other sub-experiments.

We discuss below how the ProtoGENI plug-in addresses each of these during experiment creation, describe the sub-experiment termination process, and provide a summary of how the subprocesses of experiment creation come together to give the final result.

4.1 Coordinating TIED and ProtoGENI Authorization

There are two areas in which the TIED ProtoGENI plug-in interacts with the ProtoGENI authorization system. These are a) mapping the TIED credentials into the ProtoGENI user and credential mechanism, and b) managing the ProtoGENI slice credentials used to allocate and manipulate sub-experiment resources.

4.1.1 Static Authorization Integration: TIED credentials to ProtoGENI user

The rights to manipulate a ProtoGENI slice are tied to user identities, and as such the mapping from TIED request to local authority will be a mapping from TIED attributes (ABAC or three-name) to ProtoGENI identity. One acquires a ProtoGENI identity by registering with the ProtoGENI project[3] on the Utah Emulab. The TIED developers have already registered. As part of this process, ProtoGENI issues the user an X.509 certificate to establish the user's identity.

A ProtoGENI user can request credentials from a Slice Manager to carry out various fine-grained operations on slices. Those operations closely follow the GENI Slice-Based Facility Architecture (SFA). The SFA includes operations to discover resources as well as to allocate and configure them.

To give a feel for the operations supported by ProtoGENI we list groupings that share credential requirements:

Name	**Functions**
Slice Authorization: GetCredential	Acquire the credential to create a slice, or to perform a set of operations on an extant slice (the requested privileges are part of the call)
Slice Information: Resolve, DicsoverResources, GetKeys	Query the ProtoGENI system for extant slices, available resources, and keys to allow user access to resources (specifically to get their registered SSH keys)
Special Slice: BindToSlice, Shutdown	Permit another user's GetCredential operations to succeed on this slice. Stop a running slice.
Component: Resolve, DiscoverResources	Component-level query of names and resources
Component: GetSliver, SliceStatus, SliverStatus	Retrieve information about allocations from this component manager
Component: RedeemTicket, DeleteSliver, DeleteSLice, SplitSliver, UpdateSliver, StartSliver	Make and edit resource allocations.

[3]The project is called “geni” though this disagrees with some documentation.

While it appears that the ability to see resources and perform operations are based primarily on credentials in ProtoGENI, these rights are rooted more strongly in the user identity. If a user can acquire a slice manipulation credential, all privileges are conferred to that credential, though it can be diluted to give other users fewer rights to an existing slice. DiscoverResources is an unprivileged operation on both slices and components, though one assumes not all users see the same results of those calls. Basically rights are conferred based on user identity, not on credentials.

Because user identity controls authorization in ProtoGENI, the plug-in must map the TIED credentials on an incoming request into a local ProtoGENI user under which to operate. Both the Emulab plug-in and DRAGON plug-in have code to effect such a mapping, and we will employ it in the ProtoGENI plug-in as well. This binding is expressed as a connection between the TIED credential set and the X.509 certificate representing a ProtoGENI user ID.

The mapping from TIED credentials to the ProtoGENI user that will carry out the requests is a static configuration, established in a configuration file. For example, all TIED users that can prove that they are vetted by GENI will allocate experiments as ProtoGENI user “geni,” while unvetted users will act as ProtoGENI user “faber.”

When operations are requested, the plug-in dynamically acquires ProtoGENI credentials to carry out the operations. Management of those credentials is discussed below.

4.1.2 Dynamic ProtoGENI Credential Management

This section describes the acquisition and management of ProtoGENI credentials through a sub-experiment's lifetime, described in Section 3.1.1.

When access to ProtoGENI is requested using a TIED AccessRequest, the plug-in must bind the needed ProtoGENI authorization information to the new TIED allocation ID passed in the request. Specifically, the requester's TIED credentials map to the appropriate ProtoGENI user identifier (as described above) which is bound to the unique TIED allocation ID.

If the AccessRequest includes a summary of required resources, the plug-in will use ProtoGENI services to confirm that the resources requested are accessible to the ProtoGENI user bound to the TIED allocationID.

Similarly any TIED services requested will be confirmed; we discuss service implementation below.

Assuming that the requirements above are met, a ProtoGENI slice credential is requested using the bound ProtoGENI user ID and the resulting credential is bound to the TIED allocation ID as well. This credential will have all the privileges necessary to create and manipulate the sub-experiment. Because all sub-experiment operations will be managed by the plug-in, there will be no need to delegate this credential.

One more piece of authorization material is bound to the TIED allocation ID at this time: an SSH key that the plug-in can use to remotely access general purpose computers in the experiment. While a separate key could be generated for each allocation, we will share one across multiple allocations. Note that the TIED user never sees this key; it is used by the plug-in used for experiment configuration. Part of that experiment configuration will include establishing TIED user access through other mechanisms.

At this point, the proper ProtoGENI user and credentials are bound to the TIED allocation ID to allow the rest of the operations to succeed. Following sections will discuss how those operations are composed and carried out. When the sub-experiment is terminated, the slice will be removed. When the access is terminated, the allocation ID and bound credentials will be destroyed.

4.2 Using TIED Topology Descriptions to Allocate ProtoGENI Resources

Resource allocation is one of the most important parts of sub-experiment creation, but often fairly simple to implement depending on the services of the testbed; all testbeds are in the resource allocation business to some extent. This section discusses how we are placing the TIED plug-in into the ProtoGENI version of the SFA control architecture now and later, how to mesh the topdl and ProtoGENI RSPEC resource models, and the details of experiment allocation.

4.2.1 ProtoGENI's SFA and the TIED Plug-in

ProtoGENI is implementing the GENI SFA where a clearinghouse acts as a global coordination point for identities, resources, and manager addresses. Slice Managers potentially coordinate resources across multiple Component Managers that actually allocate resources. The Slice Managers do most of the credential management while the Component Managers deal with the initial configuration of resources, including virtualization.

In principle, being a registered user of Utah's ProtoGENI site enables us to use resources across multiple associated ProtoGENI installations represented by separate Component Managers using the Slice Manager at Utah. In practice, the code is new and advertised as a little shaky.

By adhering to the published Slice Manager and Component Manager interfaces, the ProtoGENI plug-in can allocate multi-testbed sub-experiments using the SFA interface. However, this interface is somewhat primitive in how well it embeds topologies and how well it deals with Component Manager failures.

Creating sub-experiments across this interface would result in recreating much of the experiment splitting and partial failure recovery that is being put in the experiment controller here in the plug-in/access controller. This is an unattractive prototyping strategy, so we will focus on a simple case initially.

Initial implementations of the ProtoGENI plug-in will restrict themselves to one Slice Manager (by necessity) and one Component Manager (to simplify design). As

ProtoGENI's tools simplify the process of dealing with multiple Component Managers, the TIED tools will make direct use of them.

Note that it is possible to site a plug-in at each site implementing a ProtoGENI SFA interface, and use TIED's experiment manager to create cross-testbed experiments.

4.2.2 Harmonizing Resource Representations (topdl & RSPEC)

ProtoGENI's RSPEC[8] is a lower-level description of testbed resources than the TIED topdl representation, which matches the use of the RSPEC as a low-level resource allocation format. RSPECs are used in three ways in ProtoGENI, with slight distinctions made in the abstractions presented. An RSPEC can act as:

- A *resource advertisement:* available resources are described at hardware level
- A *resource request: a* subset of resources for use is specified. See below.
- A *manifest:* a description of allocated resources, including dynamic configuration decisions.

The ProtoGENI RSPEC basically breaks down into a nodes and links model of networking. ProtoGENI nodes include both general purpose computers and network infrastructure such as switches and firewalls. Though nodes are fundamentally physical resources, there are conventions for requesting VMM instances within a node. Connections between nodes and the wide-area Internet or between nodes using Cisco GRE tunnels can also be represented.

RSPECs tend to specialize their nodes through type fields and similar notation, where topdl would encourage sub-classing. Neither the syntax nor the model presents great challenges in translation, though.

Request RSPECs are versions of RSPECs that allow interconnections to be described more abstractly. The difference is between a request that specifies three computers should be connected using specific interfaces to specific ports of one or more switches that share a VLAN tag, and a request that three nodes share a virtual LAN.

ProtoGENI exports a slice embedding service[9] that takes request RSPECs with virtual connectivity requirements along with an advertisement RSPEC representing candidate hardware. That service returns a detailed request RSPEC including the full hardware-based layout that can be directly presented to a Component Manager and realized. This service currently only works across resources controlled by one Component Manager. The single Component Manager limitation is a strong motivation behind our decision to limit plug-ins to single Component Manager operation.

The TIED plug-in has to generate RSPECs from topdl at two places: when an AccessRequest includes a resource summary and when a sub-experiment is started on the ProtoGENI plug-in. At AccessRequest time, the topdl request and the available resources RSPEC must be compared to determine if the former can ever

be embedded in the latter. This does not require one to produce a best embedding, just an possible embedding. When the full sub-experiment starts, a full embedding is essential.

In both cases we plan to convert the topdl representation into an RSPEC and use the slice embedding service to find valid allocations. If during the translation process, we are unable to convert the topdl into an equivalent RSPEC, that request will fail.

The conversion of topdl to RSPEC will make use of current code that converts topdl into ns2 for Emulab-based testbeds. The ProtoGENI and Emulab resource models are very similar when an abstract request RSPEC is being created. The differences are only a matter of syntax, and the RSPEC format is well documented.

4.2.3 Creating a Sub-experiment

When the plug-in receives a StartSegment request from the experiment controller, it must use the ProtoGENI credentials created in the RequestAccess operation (described above) to allocate and configure the resources as a sub-experiment, described in Section 3.1.2. This section deals with allocating the resources and configuring them as a local experiment. We discuss the creation of inter-sub-experiment connectivity and services below. Note that a sub-experiment is not completely configured until the connectivity and services are initiated.

The topdl description is converted into a request RSPEC as described above, and the list of available resources acquired from the Component Manager using the ProtoGENI credentials and user identification mapped to the TIED allocation ID in the request. Then the ProtoGENI slice embedder is used to map the request into an embeddable RSPEC on the available resources, again using the local ProtoGENI credentials. If no embedding is found, the StartSegment call fails.

With embeddable RSPEC in hand, the plug-in proceeds to create a ticket – the SFA's promise of resources – from the Component Manager and then to redeem that ticket, which allocates the actual resources. The slice is then started. These calls are straightforward invocations of the ProtoGENI SFA with credentials and RPSECs generated above. Of course, if either of these requests fails, the sub-experiment startup fails.

It is worth noting that when a ticket is redeemed, a set of SSH keys can be specified that are placed on the general purpose computing nodes, allowing later configuration of those nodes. These keys are used below in configuring services and connectivity in the experiment.

At this point in experiment creation, the interactions with the SFA interface are complete, and experiment resources are laid out and locally connected. Now the plug-in must begin stitching the sub-experiment into its place in the federated experiment.

4.3 Using ProtoGENI Resources To Interconnect With TIED Sub-experiments

Current plug-ins support two connectivity mechanisms: best effort Internet connectivity and guaranteed service via DRAGON/OSCARS. When best effort connections are created, a gateway element is included in each sub-experiment to encapsulate local packets at a low level and tunnel them to the peer testbed. DRAGON interconnections provide the encapsulation themselves, so the traffic is tunneled directly from experiment elements. The experiment controller knows which of these are appropriate based on the global experiment topology and the advertised capabilities of the testbeds. It adds elements to the sub-experiments as needed. These elements and interfaces are marked with attributes in topdl indicating their use in connectivity.

Global experiments are split between testbeds by splitting topdl substrates; topdl elements are fully realized within one testbed. Currently the experiment controller expects the user or user's tool to have made that split. Work is ongoing in providing more general tools to do it.

The plug-in establishes connectivity described in the topdl using a small amount of local topology modification and significant configuration. We describe the process to be used in the ProtoGENI plug-in to implement the two mechanisms above and suggest short term extensions.

4.3.1 Best Effort Connectivity

The ProtoGENI plug-in will initially provide best effort connectivity using SSH tunnels at the link level. SSH is used because it uses a nearly universally available service to provide basic functionality. Relying only on such a basic system allows us to support as many testbed technologies as possible.

The experiment controller places gateway elements in the local experiment's copy of the substrate that is split between the two experiments. When the sub-experiment description is converted into an RSPEC, the gateway nodes are connected to the Internet.

After the ticket is redeemed, the plug-in uses its remote access to configure the node for link-level SSH forwarding. This process includes the following:

1. Allow remote access from the other gateway node by placing SSH public keys into the appropriate authorization files. Keys for this are passed by the experiment controller to the plug-in.

2. Load or enable any required kernel modules to support bridging or link level forwarding. Our current implementation loads bridging and tunneling support if it is not already available. These modules can be acquired from the experiment controller using the mechanism described in Section 4.4.2.

3. Configure local routing or interfaces for Internet access. The ProtoGENI plug-in will establish source routes to the peer and little else.

4. The active gateway will establish the SSH tunnel between nodes and bridge the appropriate interfaces onto it on both sides.

As step 4 implies, the experiment controller designates one of the gateway elements as the active initiator of the connection. When the tunnel is created, the active side picks the interface name for both sides of the tunnel, and then remotely connects that tunnel interface to the appropriate physical interface. There is no point in the passive end doing some of that work before the connection is made, and resolving any naming conflicts in the tunnel interfaces is simplified by having the active side pick them.

Our current implementation of this tunneling system is implemented on FreeBSD and uses its vernacular for the tunnel interface names and other features. ProtoGENI does not currently allow one to specify the operating system installed on a computer, so we will need to port this code to the operating system that they export (a Linux distribution). The tools are all off the shelf components, so this work will be straightforward.

4.3.2 Dedicated Connectivity

The implementation of direct connectivity is very simple in ProtoGENI.

Guaranteed connectivity services generally provide the link level encapsulation and other features that gateway nodes provide without the need to add additional gateways to the sub-experiment. These are usually connected to a testbed at a well-defined point. Often this is a particular VLAN tag or port on a switch.

In Emulab testbeds, the difficulty is piercing the node/network abstraction to connect the experiment elements to the physical hardware providing guaranteed service. In ProtoGENI, this is greatly simplified by the direct access to the hardware that the RSPEC provides. The request RSPEC is doctored to include the appropriate hardware connections when the topdl is converted into the request RSPEC, before it is presented to the slice embedding service. The plug-in knows how to do this as the elements are marked in the topdl and the hardware point is local configuration state of the plug-in.

However, a stumbling block here is that DETER/TIED and Emulab/ProtoGENI do not share either a ProtoGENI backbone access point or a DRAGON access point, making it difficult to test this connectivity. Both groups are working to resolve the political and logistical issues in getting the two testbeds on one wide area service offering provisioned access, and this design will support it when available.

4.3.3 Other Connectivity Extensions

There are some other connectivity possibilities that are suggested by the ProtoGENI system that are not in the initial development plan, but will be explored as time and availability merit. Currently we are considering Cisco GRE tunnels and OpenVPN encapsulation.

Cisco GRE tunnels are supported directly by ProtoGENI between components that are connected to appropriately enabled equipment. Though DETER and other

affiliates do not provide direct access to this encapsulation mechanism, it is a distinct possibility to add it. It would be exported by a gateway node instead of the least-common-denominator SSH tunnel.

Similarly, OpenVPN can be used to create VPNs in the wide area. OpenVPN is attractive because of its more sophisticated model of a VPN. SSH treats the VPN as a simple tunnel over TCP, while OpenVPN supports unreliable forwarding and connection reestablishment in the face of failures. It can export the same tunneling interface (i.e, an `tun0` network interface) as SSH does, which would make a port straightforward.

The primary issue is that OpenVPN is included in the core of fewer operating systems than SSH, though the advantages may justify installing it dynamically.

4.4 Establishing TIED Experiment Services Using ProtoGENI Resources

Though there are numerous differences between an Emulab and ProtoGENI, perhaps the greatest difficulty in creating a TIED environment is how few experimental services ProtoGENI nodes have by default. The experiment nodes are pretty close to being vanilla Linux installations.

This different service environment is one of the motivations for our modular service interface. Rather than predicate services on exporting an Emulab environment, we will be providing individual services. The first set of services implemented in the ProtoGENI plug-in will be those that were originally part of the Emulab project export, though they will be more generally implemented and factored. We break these down into configuration services and traditional services.

4.4.1 Configuration Services

Configuration services provide the bedrock configuration information that allow the plug-in to configure resources so researchers can access them. The ProtoGENI SSH key seeding as part of redeeming a ticket is a simple version of these kind of service. Our plan is to provide a simple service that provides an initial access environment appropriate to the element. In terms of ProtoGENI, for now, this boils down to initializing a set of Linux accounts on each node compatible with the traditional services we define below.

Unlike the traditional services, configuration services are accessed once by the plug-in when the experiment is created. This sets the stage for the traditional services to operate.

We factor configuration services out of the experiment controller to make the configuration of experiment orthogonal to the topology creation. The same set of user accounts can be used across multiple topologies or instances of the experiment.

Our configuration service will provide the following information to the plug-in, which will use a subset of it appropriate to the element being configured. For example, a

switch may just have SSH keys added to its set of authorized keys, while a Linux-based computer may use all of the information directly.

Name	**Function**
Groups	Standard Unix characterization of a group. Name, numeric identifier, members.
Users	Standard minimal Unix characterization of a user with access extensions. Name, numeric ID, password, home directory, command interpreter, SSH public keys, X.509 certificates.
Mount points	Characterization of any shared file system to be used. Where to place files systems in the standard hierarchy, the file system to use, and other common options.
Host map	Name ↔ IP address mappings for key hosts

If the configuration service is present in the list of services exported to the sub-experiment, the ProtoGENI plug-in will contact that service using a bidirectionally authenticated SSL connection and retrieve the configuration information. When the sub-experiment resources are allocated, the plug-in will configure the nodes appropriately using that data. In particular, it will create the necessary groups and user accounts on the local Linux-based machines, using the access granted by the RedeemTicket call.

The experiment controller will have negotiated access to the configuration service for the sub-experiment access controllers that need it.

4.4.2 Traditional Services

Providing the more traditional services is primarily a matter of forwarding connections to the systems providing the service using service gateway elements. A service gateway element is an analog to the connectivity gateway portals. Rather than providing link-level forwarding, these nodes forward packets to remote services. In practice, service gateways and connectivity gateways often physically share a node.

Most of the services we forward are TCP services that can be directly tunneled using SSH or similar systems. Some testbeds, like DETER, do not assign any globally routeable addresses to experiment nodes. In such cases, the service gateways are

necessary to allow packets outside the network at all. In other cases, firewalls or other interpositions require their use.

While server/client port forwarding has carried us a long way, we expect that some services will require more general network address translations. This work is ongoing.

In the ProtoGENI plug-in, sub-experiments that import or export traditional services will have service gateway nodes allocated to them by the experiment controller. Before services are started on the sub-experiment nodes, the service gateway node will be configured (using the SSH access) to forward the configured services. This may include routing configurations similar to establishing a connectivity gateway. Once a service gateway is configured, other experiment nodes can begin accessing services.

Providing remote access to service providers is only half the problem of configuring services on ProtoGENI computers. Some services configured for the experiment may be unavailable on the standard ProtoGENI installation, and the plug-in is responsible for installing and configuring those services.

On order to make such configurations, first the plug-in must gather the software necessary for the service. The experiment controller will make such software available to the plug-ins through a secure repository. This repository uses the TIED identifier and credentials of the plug-in to control access to the service software repository. This repository and software acquisition protocol is already implemented in the Emulab plug-in, and its incorporation into the ProtoGENI plug-in will be straightforward. This software acquisition is accomplished before the ticket is redeemed.

Once the various service software has been acquired, the plug-in must install it on the nodes that require it. In many cases this is straightforward, accomplished by installing an RPM or tarfile. There is an extension to the request RSPEC to install such software. In other cases it may be more complex. For example, using a remote filesystem that is not compiled into the kernel. For some services, the overhead of installing the service will be high enough that the plug-in will be configured to disallow it.

To summarize, the ProtoGENI plug-in will go through three steps in starting a traditional service:

- Acquire software for services that are not supported in the basic ProtoGENI image from the experiment controller
- Configure and start the service gateway node. It will forward connections outside the testbed to the service provider.
- Install, configure, and start services on experiment nodes

4.4.3 Initial Services Supported

The services supported by the initial ProtoGENI plug-in will be a factorization of the existing Emulab project services. Specifically the following will be supported:

Name	**Description**
Configuration service	Export of user/group/hostname environment.
SMB filesystem	Shared filesystem that spans experiments.
SEER experiment support and event system	DETER experiment control system that provides experiment monitoring and an extensible event system.

These services are chosen because they represent a proven basis for a useful experiment environment as well as a range of implementation complexity.

4.5 Termination of Sub-Experiments and ProtoGENI Access

When an experiment is terminated, the ProtoGENI DeleteSlice operation is called. Should it fail, the Shutdown operation is invoked. At this point the credential bindings remain in place. They can be used to populate an new slice without making additional calls.

When the TIED RemoveAccess call is made, the credentials are unbound an destroyed, as is the TIED allocation ID.

4.6 Plug-in Creation Control Flow

The preceding sections each focused on an aspect of sub-experiment creation to make the challenges and implementation strategies clear. With a picture of each subsystem in place, we walk through the entire slice creation flow of control.

- **TIED RequestAccess**
    - TIED credentials are mapped to ProtoGENI user ID (or fail).
    - ProtoGENI slice credential acquired from Slice Manager and bound to TIED Allocation ID (the ID is in the request).
    - If resource summary is included in request, translate request to Request RSPEC and confirm resources exist using ProtoGENI Slice Embedding Service (or fail).

- If service summary is included, confirm against static list of supported services (or fail).
- SSH key for configuration bound to TIED Allocation ID.

- **TIED StartSegment**
  - Convert topdl description to abstract request RSPEC.
    - If direct gateways appear, link them to proper hardware in RSPEC.
    - if best effort gateways appear, connect them to Internet in RSPEC.
  - Use ProtoGENI Slice Embedding Service to get realizable request RSPEC.
  - Allocate resources to sliver (using credential bound toTIED allocation ID) using GetTicket (or fail).
  - Realize Resources using RedeemTicket and StartSliver (or fail).
  - Retrieve service software from experiment controller.
  - If configuration service is specified, get configuration information.
  - Configure connectivity gateways (if any) and start them (Stra.
  - Configure service gateways (if any).
  - Configure services on experiment nodes and restart the services.
- **TIED StopSegment**
  - Call StopSlice on associated sliver (credential bound to TIED allocation ID) and Shutdown if necessary
- **TIED RemoveAccess**
  - Delete ProtoGENI credentials
  - Delete TIED allocation ID

5 Summary

This document presents a functional specification for TIED plug-ins, including detailed descriptions of the data flow and message contents that a new plug-in must support. The document also describes the topology description language currently in use by TIED plug-ins, outlining the important representations and extension methods.

In addition to textual descriptions contained in the document, links are provided to on-line, detailed specifications for plug-in messages, interfaces, and the topology description language.

References

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

[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]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).

[4] ns-2, *http://nsnam.isi.edu/nsnam/index.php/Main_Page* .

[5] John K. Ousterhout, "Tcl: An Embeddable Command Language," *USENIX Conference Proceedings,*Washington, D.C., (January 1990).

[6] 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).

[7] Chin Guok, David Robertson, Mary Thomposn, Jason Lee, Brian Tierney, and William Johnston, "Intra and Intedomain Circuit Provisioning Using the OSCARS Reservation System,*" in Proccedings of IEEE Broadban, Connections, and Systems (BROADNETS 2006),* (October 2006).

[8] ProtoGENI RSPEC, *http://www.protogeni.net/trac/protogeni/wiki/RSpec* .

[9]ProtoGENI Slice Embedding Service, *http://www.protogeni.net/trac/protogeni/wiki/SliceEmbeddingServiceAPI* .