PlanetLab Implementation of the Slice-Based Facility Architecture

Larry Peterson, Princeton
Soner Sevinc, Princeton
Scott Baker, Arizona
Tony Mack, Princeton
Reid Moran, Princeton
Faiyaz Ahmed, Princeton

Draft Version 0.07

September 8, 2009

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

Table of Contents

1 Introduction

PlanetLab supports a prototype implementation of the abstractions and interfaces defined in the *Slice-based Facility Architecture* (SFA) document. This paper outlines a PlanetLab-centric "projection" of the SFA, and provides details about the implementation.

PlanetLab Central (PLC) bundles together an aggregate manager, a slice manager, and a registry server. Individual PlanetLab nodes correspond to components and run a component manger. The PLC aggregate, the PLC slice manager, and the component manager running on each node export the slice interface.[1] The PLC registry server exports the registry interface.

The PlanetLab Consortium serves as a top-level slice and management authority. Sub-authorities correspond to member institutions, as well as federated partners. For example, plc.princeton.codeen is the human-readable name for the CoDeeN slice from Princeton, plc.vini.internet2.nyc.node1 is the HRN for a component in the VINI backbone, and plc.eu.inria is the HRN of a slice authority within the PlanetLab Europe sub-authority.

Throughout this document it is important to keep in mind the distinction between the "public" PlanetLab, which is administered by Princeton and runs an instance of PLC at **`http://www.planet-lab.org`**, and other "private" instantiations of PlanetLab that take advantage of the MyPLC software distribution. These other instantiations are able to set up their own aggregates and registries, and federate them with the public PlanetLab. The goal is for this federation of aggregates to share a common name space and interfaces—e.g., plc.princeton.codeen is a universally recognized slice name and all aggregates support a CreateSlice operation—where each member aggregate establishes an allocation policy for its own resources.

2 Engineering Decisions

As a working system, PlanetLab has made certain engineering decisions. This section outlines these decisions and their implications on the SFA.

- PlanetLab maintains all authoritative state at PLC. Individual nodes maintain only cached state that must be updated should the node fail and subsequently reboot. This means, for example, that any RedeemTicket or LoanResources operations invoked on a node must be re-invoked whenever the node reboots. Note that each node does have persistent storage that records certain information for the slices it hosts (e.g., the fact that the slice exists and is mapped to a particular virtual machine), but this state may become out-of-date during the time a node is down. Also, the node may be reinstalled, which clears all local state.
- Nodes implicitly delegate control over their resources to PLC (the aggregate), which is responsible for implementing PlanetLab's resource allocation policy. As a consequence, the GetTicket, CreateSlice, and UpdateSlice operations succeed on PLC, but fail when

[1] The GENI literature refers to a Clearinghouse, which can be viewed as a bundle of related software packages—e.g., an aggregate manager and registry server—and a "trust anchor." PLC can be viewed as an example GENI Clearinghouse on both counts.

invoked on individual nodes. Technically, these per-node invocations are return a "no available resources" message in response to requests to allocate resources since they have relinquished the right to allocate their resources to PLC. Individual nodes do, however, support the RedeemTicket and LoanResources operations, so it is possible to get a ticket from PLC and then redeem it on individual nodes. Both PLC and individual nodes support all other operations defined by the slice interface.

- The slice manager running at PLC is configured to know about one or more aggregates. By default, it knows about the local PLC aggregate, but through peering arrangements with other management authorities, it can be configured to provide users with an interface to multiple aggregates. Like an aggregate, the slice manager exports the slice interface, the only difference is that the slice manager does not support the GetTicket operation; slices can be created only using CreateSlice. Users that want to retrieve and redeem tickets must contact individual aggregate managers and their components. A set of helper functions, which runs on the user's machine and not at PLC, augments the slice interface to provide researchers with a richer interface for manipulating slices.
- Tickets are idempotent. This means no matter how many times one redeems a ticket granting a slice 1Mbps of link bandwidth, for example, the slice is granted only 1Mbps of link bandwidth. In other words, tickets specify absolute resource capacity, rather than relative or incremental capacity. On the other hand, the LoanResources operation does increment a slice's resource allocation by the amount given in the RSpec.

PlanetLab's current resource allocation policy is fairly simple. Slices are granted "best effort" resources by default. The policy recognizes only select slices as qualifying for guaranteed resources. One of these corresponds to the Sirius Reservation Service, which subsequently uses the LoanResources operation to grant other slices link and CPU guarantees for one-hour time slots. Note that each aggregate is free to define its own resource allocation policy, that is, what slices it is willing to host and how many resources to grant each of those slices.

PlanetLab supports an extensive O&M interface that goes well beyond anything defined by the SFA. This is a private interface known only to PlanetLab operators. One can view the SFA management interface as a small subset of this PlanetLab-specific O&M interface that is common to all components participating in a federated slice-based facility.

3 Usage Scenarios

This section walks through a sequence of usage scenarios showing how we expect PlanetLab to evolve to take advantage of the SFA to support both federation and third-party user services. Throughout this section, we use the notation outlined in Figure 3.1.

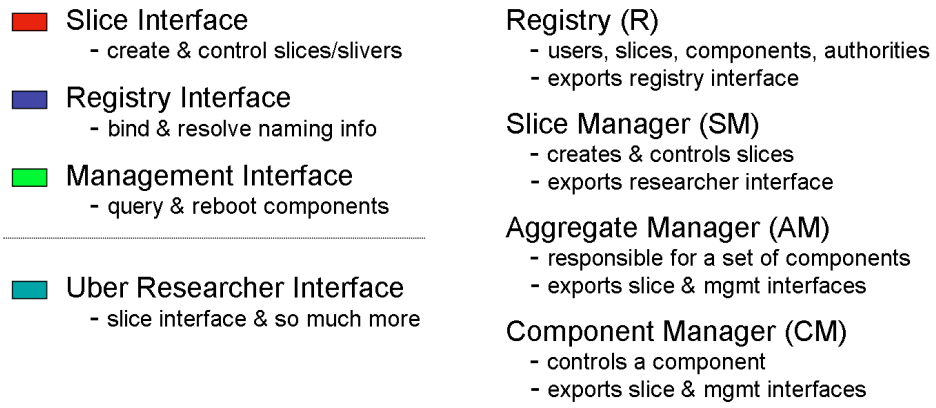

Figure 3.1: Notation used throughout this section, including both interfaces and managers.

For the purpose of this discussion, we introduce an *uber researcher interface,* which provides a high-level interface (possibly GUI-based) that researchers interact with to set up, control, and tear down their slices. This interface is not one of the standard SFA-defined interfaces, although it likely extends the slice interface. For example, it might allow users to manipulate graphical representations of their slices, it might iteratively discover and acquire resources, and it might help users steer the experiments running in those slices. In the current implementation, the researcher interface corresponds to a combination of the slice interface exported by the slice manager and the set of helper functions running on the researcher's desktop.

3.1 Single PlanetLab

The first scenario, depicted in Figure 3.2, corresponds to a simple deployment of PlanetLab, in which a trivial slice manager (SM), a single aggregate manager (AM), and a registry (R) are all bundled in PLC, with each node running a component manager (CM).

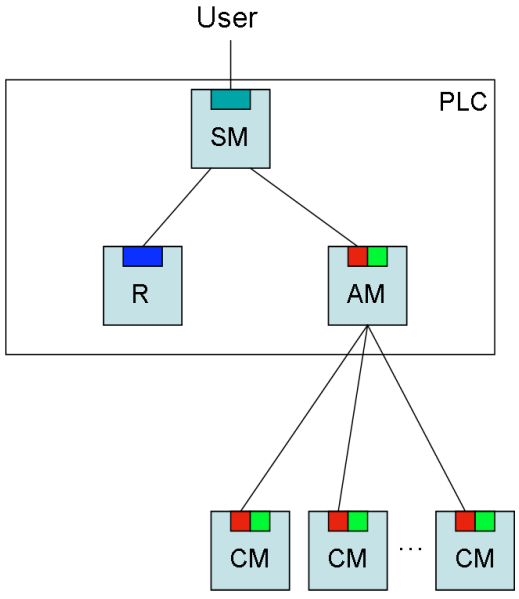

Figure 3.2: Single PlanetLab, with bundled slice manager, registry, and aggregate manager.

In this example, users interact with the slice manager (using either a GUI or a programmatic interface) to create and control their slices. The slice manager contacts the registry to retrieve the necessary credentials, and then invokes the slice interface on the aggregate to create and control the slice. As is the common case in PlanetLab, the aggregate (rather than end users) interacts with the individual nodes. Note that the current implementation of PLC uses a private interface to interact with the individual components (although the components also export the slice interface to other clients).

In all the examples presented throughout this section, we focus on the slice-related records in the registry. Component-related records are also recorded in the registry, but we do not illustrate how these records are used in the following discussion. Currently, PLC manipulates these records on behalf the constituent components, with PLC and the components communicating using a private interface.

3.2 Multiple Aggregates

The scenario depicted in Figure 3.3 spans two aggregates—PlanetLab and VINI—each responsible for its own set of components. That is, VINI and PlanetLab are distinct management authorities, each defining its own policies for what slices they are willing to host and how many resources to grant each slice. In this case, VINI does not operate its own registry or slice manager, and PlanetLab's slice manger presents users with a unified view of all the components available on both systems, hiding the fact that its global view spans multiple aggregates and giving users what effectively amounts to "single sign-on".

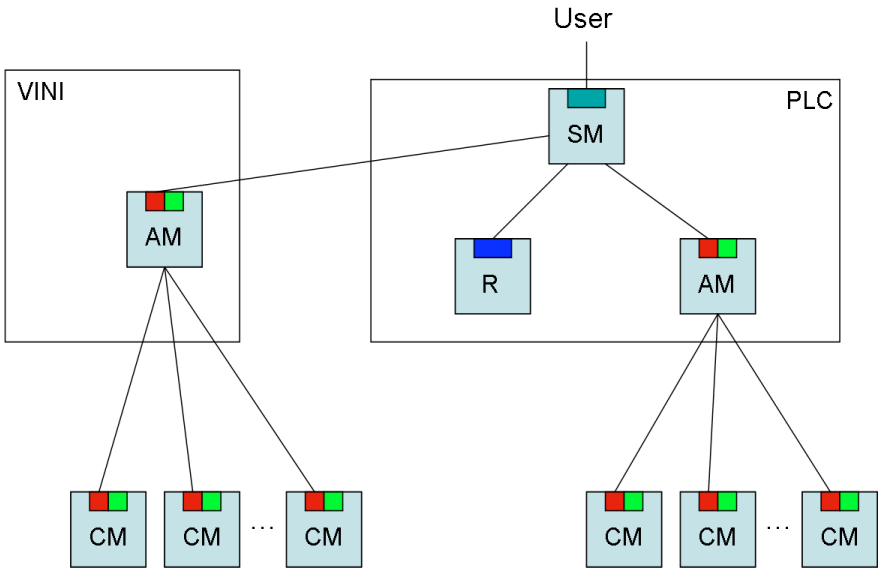

Figure 3.3: VINI and PlanetLab represent independent aggregates (and corresponding management authorities), unified by a single slice manager.

To create a slice, the PlanetLab SM would need to contact both available aggregate managers to learn about the available components. It would then present these components to the user in an SM-specific way. Once the user selects the set of components to be included in his or her slice, the SM would call the Registry to retrieve the necessary credentials, and then invoke the CreateSlice operation on the respective aggregates to create the cross-aggregate slice.

In this scenario, the SM plays the role of an *aggregate of aggregates*. When viewed from this perspective, it makes sense that the SM exports the slice interface, just like any other aggregate (i.e., the uber researcher interface is a superset of the slice interface). Note that there are several candidate aggregates that we plan to include in an early demonstration of this capability—VINI, Measurement Lab (M-Lab), Enterprise-GENI, the Mid-Atlantic eXchange (MAX), GpENI, and a Supercharged PlanteLab Platform (SPP) network embedded in Internet2.

3.3 Full Federation

Our final scenario, shown in Figure 3.4, involves symmetric federation between two autonomous aggregates (or sets of aggregates), one representing PlanetLab Europe (PLE) and the other representing the rest of PlanetLab (PLC). Both systems support their own slice manager, registry servers, aggregate manager, and set of components. As in the previous scenario, users interact with their "local" SM, which creates and manages slices spanning both aggregates.

Although not explicitly depicted in the figure, the PLC registry points to the PLE registry. That is, registry records for the top-level PlanetLab authority, including the record for the EU sub-authority, are maintained in the PLC registry, while records associated with the EU sub-authority are maintained in the PLE registry server.

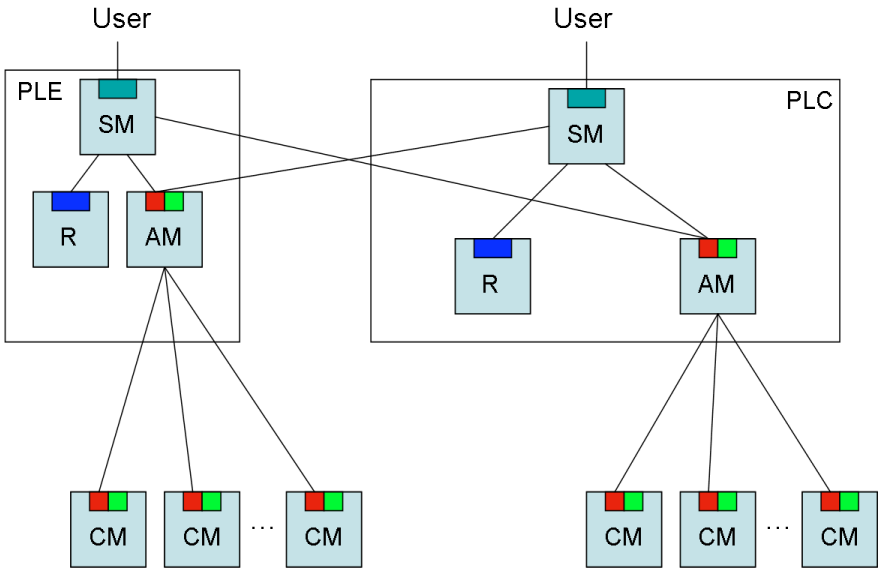

Figure 3.4: Peer testbeds (PLC and PLE) federate their aggregates.

Note that while Figure 3.4 depicts each SM as directly accessing the single AM available in the peer, in general, each peer SM might know about a set of local aggregates, where each SM accesses the peer aggregates indirectly through the peer's SM (which is acting as an aggregate of aggregates). When viewed in this way, the SMs are analogous to tier-1 ISPs in today's Internet—where each SM knows about all other SMs—with the analogous tier-2 and 3 AMs accessed indirectly through their tier-1 SM.

There are four candidate peers that we plan to include in an early demonstration of this capability—PlanetLab Europe (PLE), PlanetLab Japan (PLJ), PlanetLab Korea (PLK), and PlanetLab Brazil (PLB).

3.4 Other Possibilities

Many other configurations are possible. For example, there may exist slice managers that are not bundled with an aggregate or set of components. Such a slice manager would provide an interface to one or more existing aggregates, or other slice managers aggregating a set of aggregates. In the limit, individual users can interact directly with one or more aggregates to acquire resources (instantiate their slices) without involving a slice manager. Or said another way, each user could run his or her own slice manager functionality. As another example, other independent aggregates might choose to trust users registered with PlanetLab—retrieving their credentials from the PLC-rooted registry—but otherwise instantiate their slices purely within their own aggregate.

4 Implementation

We have implemented a module, called **`sfa`**, that exports the set of interfaces defined by the Slice Facility Architecture. The **`sfa`** module is distributed as part of the MyPLC software package, and is independently available at **`http://svn.planet-lab.org`**. The module code is organized as follows:

`sfa/client`: User-level programs and tools used by researchers and authorities to access a federation of sliced-based testbeds. Primary among these is **`sfi.py`** (slice facility interface), which is described in more detail in the appendix.

`sfa/server`: First-class objects involved on the server side, corresponding to the three main classes that implement the Aggregate Manager (AM), Slice Manager (SM), and Registry (R).

`sfa/plc`: PLC-specific code underlying the server-side software. Of particular note, **`sfa-import-plc.py`** imports a PLC database and produces an SFA database.

`sfa/trust`: Classes defining the basic identification and authentication objects, including credentials.

`sfa/methods`: Classes defining the individual methods that make up the Registry and Slice interfaces.

`sfa/util`: Assorted utility programs.

4.1 Certificates, Credentials, and GIDs

A **`Certificate`** class extends pyOpenSSL's native X.509 certificate class by adding a parent field, thereby supporting certificate chains. When loaded from a file or a string, a sequence of certificates is interpreted as a parent chain. When saved to a file or string, the caller can choose whether or not to save the chain of parent certificates.

The **`sfa`** module also defines **`GID`** and **`Credential`** classes that extend this **`Certificate`** class. They add support for GID-specific and credential-specific tuples, respectively. The **`GID`**

class sets the **subject-public-key** field of the certificate to the PublicKey in the GID, and the **subject-alt-name** field of the certificate to the UUID and HRN in the GID. UUIDs are generated according to RFC4122 (version 4). The authority that is responsible for the object denoted by the GID signs the certificate. Similarly, the **Credential** class stores the entire (GIDCaller, GIDObject, LifeTime, Privileges, Delegate) 5-tuple in the **subject-alt-name** field. Both GIDs and credentials currently use 1024-bit RSA as their public key algorithm and SHA1 as their signature algorithm.

4.2 RSpec

The definition of the **RSpec** class is rapidly evolving. More information will be posted soon.

4.3 Tickets

Similar to GIDs and credentials, a **Ticket** class extends the **Certificate** class. The implementation deviates somewhat from the definition given in the SFA document. A ticket is defined by the 5-tuple (GIDCaller, GIDObject, Attributes, RSpec, Delegate), where GIDCaller is the GID of the principal performing the operation; GIDObject is the GID of the slice to which the ticket is bound; Attributes is the set of PlanetLab attributes (tag/value pairs) that do not correspond to resources (e.g., **keys** contains the SSH keys for the users affiliated with the slice, **vsys** specifies privileged operation the slice may invoke, and initscript defines the script that runs when the slice boots); and RSpec specifies the set of resources bound to the slice.

As described in the Section 2, PlanetLab allows callers to get tickets from the Aggregate Manager running at PLC, and redeem tickets at the Component Manager running on each node. Individual nodes do not respond to local calls to GetTicket.

4.4 XML-RPC

Secure remote invocation is based on XML-RPC running on top of HTTPS, where the underlying SSL layer is implemented using pyOpenSSL. Both the client and server specify their private and public keys when opening an SSL socket, and upon successful connection establishment, each knows the other's public key (by convention, this key is stored in an X.509 certificate). The **GeniClient** and **GeniServer** classes implement the underlying client and server, respectively (both are in the **/sfa/util** directory).

The next step of XML-RPC is to dispatch the incoming call to the specified operation, where the first argument to each operation is a credential. The **Auth** class (in **trust/auth.py**) provides a **check** method to verify that this credential gives the caller the right to invoke the corresponding operation. For example, the following code snippet is used by all SFA methods:

```
def sfa_method(self, cred_str, ...other arguments...):
   self.api.auth.check(cred_str, "sfa_method")
   ...
```

The **check** routine is defined as follows:

```
##
# Check the credential against the peer cert (callerGID included
# in the credential matches the caller that is connected to the
```

```
    # HTTPS connection, check if the credential was signed by a
    # trusted cert and check if the credential is allowed to perform
    # the specified operation.

    def check(self, cred_string, operation):
        # extract relevant fields from the credential
        self.client_cred = Credential(string = cred_string)
        self.client_gid = self.client_cred.get_gid_caller()
        self.object_gid = self.client_cred.get_gid_object()

        # make sure client_gid is not blank
        if not self.client_gid:
            raise MissingCallerGID(self.client_cred.get_subject())

        # make sure pubkey of peer matches pubkey in client_gid
        peer_cert = self.server.peer_cert
        if not peer_cert.is_pubkey(self.client_gid.get_pubkey()):
            raise ConnectionKeyGIDMismatch(
                                self.client_gid.get_subject())

        # make sure client is allowed to perform the operation
        if not self.client_cred.can_perform(operation):
            raise InsufficientRights(operation)

        # make sure credential is signed (recursively) by parents
        if self.trusted_cert_list:
            self.client_cred.verify_chain(self.trusted_cert_list)
            if self.client_gid:
                self.client_gid.verify_chain(self.trusted_cert_list)
            if self.object_gid:
                self.object_gid.verify_chain(self.trusted_cert_list)
```

where **verify_chain** recursively walks through the chain of certificates, making sure that each parent signed the certificate of each child. At each level of recursion, **verify_chain** makes sure that the parent's HRN is a prefix of the child's HRN. The calls to **verify_chain** at the end of **decode_authentication** conservatively verify the credential, the client's GID, and the object's GID.

4.5 SFA Interfaces

The **sfa** module implements classes for each of the major objects described in the SFA: Registry (co-located with PLC), Aggregate Manager (co-located with PLC), Slice Manager (co-located with PLC), and Component Manager (co-located with each node). This section briefly describes the implementation of each.

Note that all managers (servers) need not to be activated on all instantiations. For example, the standard PLC instantiation includes a Registry (for all authorities, slices, users and components managed by this PLC), an AM (for the aggregate of nodes managed by this PLC), and an SM (which provides users with an interface to the set of aggregates that peer with this PLC). On the other hand, a VINI or M-Lab instantiation runs only an AM. The **sfa-server.py** command is executed to start up the desired set of servers, and depending on the command-line arguments,

instantiates the requested combination of a **`Registry`** object, an **`Aggregate`** object, and a **`SliceMgr`** object.

Also note that while the **`sfa`** module is usually bundled with the MyPLC distribution, it can be accessed in isolation, for example, to provide a starting point for building a stand-alone AM or CM. When used in this way, the PlanetLab-specific code would be replaced with software that knows how to control the devices and networks of interest. That is, the developer would be using only the secure remote invocation machinery implemented by **`sfa`**.

4.5.1 Registry

The **`Registry`** class exports exactly the registry interface defined by the SFA, where the **`Record`** class defines a registry record; it includes the 4-tuple (Name, GID, Type, Info). The class is implemented on top of the PlanetLab database, where the Info field contains the following three sub-fields: pointer, pl_info, and geni_info. The first sub-field is a pointer into the PlanetLab database, its exact representation depending on the record Type (e.g., if Type==user, then the pointer is a person_id that indexes PlanetLab's persons table). The pl_info and geni_info sub-fields represent two different views of the information maintained by the registry for the named object. The pl_info sub-field is a cache of information read from the PlanetLab database about the named object, and the geni_info sub-field a mapping of that information into "SFA canonical form." Today that mapping is implemented by the identity function. In the near term, we plan to generate code to parse and verify registry records from an EMF-based schema, just we do today for RSpecs.

4.5.2 Aggregate Manager

The **`Aggregate`** class exports exactly the slice interface defined by the SFA. It is implemented on top of PlanetLab's PLCAPI, and as such, is used to create, terminate, and control slices on a PlanetLab-wide basis. It can also be used to retrieve a ticket that is subsequently passed to the CM running on any PlanetLab node managed by this instance of PLC. Note that there is a different **`Aggregate`** class for each supported aggregate. For example, the VINI-specific aggregate understands how to act on the topology-related information in an RSpec.

4.5.3 Slice Manager

The **`SliceMgr`** class exports the slice interface, and in turn, calls the slice interface on the set of aggregates with which this instance of PLC peers. The SM calls each peer AM to learn the set of available components (and subsequently displays the union of these lists to users), and later calls the appropriate AMs to instantiate a slice on the components managed by that AM. The SM maintains a database of all slices created by behalf of users, including a record of where those slices have been instantiated to it can "pass through" slice control operations. The SM does not support the ticket-related calls; clients that want to retrieve a ticket must directly contact an AM.

4.5.4 Component Manager

The **`Component`** class exports exactly the slice interface defined by the SFA. It is implemented on top of PlanetLab's Node Manager (NM) interface available on each node. Currently, slice control operations invoked on a CM are successful only if the slice was created using a ticket.

Control of slices instantiated by invoking CreateSlice on the PLC aggregate must go through the PLC aggregate.

4.6 Command-Line Interface

Users invoke the **`sfi`** command to manage their slices. **`sfi`** is configured to interact with a "home" registry and slice manager to implement those commands. **`sfi`** does not directly interact with any aggregates.

`sfi` manages a set of credentials on behalf of the user, and uses them when invoking various slice or registry operations. The set of credentials include a "user" credential (used to view information in the registry), a "slice" credential for each slice that the user belongs to (used to create, control, and terminate the slice), and if the user also serves as PI for a research organization, an "authority" credential (used to register nodes, slices, and users in the registry).

In addition to **`sfi`**, we are developing a set of tools that make it easy for users to manipulate RSpecs and registry records, the two key file formats employed by **`sfi`**. (**`sfi`** uses a working directory, **`~/.sfi`** by convention, in which it stores credentials, RSpecs, and registry records.) These tools are still under development, but **`/client/editRecord.py`** is an example of a simple tool that can be used to edit a file that stores a registry record.

Appendix: User Tools

This appendix describes the available user-level tools and commands using a Unix manual-like template.

NAME

sfi – Slice Facility Interface

SYNOPSIS

```
sfi [options] command [command-options] [command-args]
```

DESCRIPTION

Provides a Unix command-line interface to a federation of PlanetLab-based networks that export the programmatic interfaces of the Slice Facility Architecture (SFA). **sfi** is configured to invoke operations on a "home" registry and slice manager. It manages a set of credentials on behalf of the user, and uses them when invoking various slice or registry operations. The set of include a "user" credential (used to view information in the registry), a "slice" credential for each slice that the user belongs to (used to create, control, and terminate the slice), and if the user also serves as PI for a research organization, an "authority" credential (used to register nodes, slices, and users in the registry).

A typical mode of operation is to retrieve a registry record (it can be saved as an XML file), edit the record file locally, and then use this modified record file to update the registry. Similarly, a user can retrieve a default rspec from the slice manager (it can be saved as an XML file), edit the rspec file locally, and then use this modified rspec file to create or update a slice.

COMMANDS

list – list registry records belonging to a named authority. If the **-t** option is present, limit the output to the selected record type.

```
sfi list [command-options] name
   -h, --help              show help message
   -t TYPE, --type=TYPE    (user|slice|ma|sa|node|aggregate)
   -o FILE, --file=FILE    output XML to file rather than standard output
```

show – output named registry record(s). If the **-t** option is present, limit the output to the selected record type.

```
sfi show [command-options] name
   -h, --help              show help message
   -t TYPE, --type=TYPE    (user|slice|ma|sa|node|aggregate)
   -o FILE, --file=FILE    output XML to file rather than standard output
```

remove – remove named record(s) from registry. If the **-t** option is present, remove only the record of the selected type

```
sfi remove [command-options] name
   -h, --help                show help message
   -t TYPE, --type=TYPE      (user|slice|ma|sa|node|aggregate)
```

add – add record in named file to registry.

```
sfi add [command-options] record
   -h, --help                show help message
```

update – update registry with record in named file.

```
sfi update [command-options] record
   -h, --help                show help message
```

slices – list all slices available via slice manager.

```
sfi slices [command-options]
   -h, --help                show help message
```

resources – output rspec for resources associated with the named slice. If the slice has been instantiated, the resulting rspec corresponds to the resources currently allocated to the slice. If the slice has not yet been instantiated, the resulting rspec corresponds to the default rspec accepted by the slice manager for that slice. If the **-f** option is present, print only DNS names or IP addresses for nodes identified in the rspec.

```
sfi resources [command-options] name
   -f FORMAT, --format=FORMAT    (dns|ip)
   -o FILE, --file=FILE      output XML to file rather than standard output
   -h, --help                show help message
```

create – create named slice according to the rspec in given file. If the slice already exists, update the slice according to the rspec. A PI must have already created a registry record for the slice before a user is allowed to invoke **create** on that slice.

```
sfi create [command-options] name rspec
   -h, --help                show help message
```

delete – delete named slice.

```
sfi delete [command-options] name
   -h, --help                show help message
```

reset – reset named slice.

```
sfi reset [command-options] name
   -h, --help                show help message
```

start – start named slice.

```
sfi start [command-options] name
   -h, --help                show help message
```

stop – stop named slice.

```
sfi stop [command-options] name
   -h, --help                show help message
```

OPTIONS

-h, --help	show help message
-r URL, --registry=URL	root registry server
-s URL, --slicemgr=URL	slice manager
-d PATH, --dir=PATH	working directory (**~/.sfi** by default)
-u HRN, --user=HRN	user name
-a HRN, --auth=HRN	authority name
-v, --verbose	verbose mode

FILES

~/.sfi – default working directory where credentials are collected

Registry records (argument **record** in the above synopsis) and resource specifications (argument **rspec** in the above synopsis) are saved to and read from XML files.

ENVIRONMENT VARIABLES

SFI_USER=HRN	user name
SFI_AUTH=HRN	authority name
SFI_SM=URL	slice manager
SFI_REGISTRY=URL	root registry server

For example

```
SFI_USER=plc.princeton.llp
SFI_AUTH=plc.princeton
SFI_SM=http://128.112.139.90:12347/
SFI_REGISTRY=http://128.112.139.90:12345/
```

NAME

getRecord, setRecord – get and set fields in registry record

SYNOPSIS

```
getRecord [name]

setRecord [command-args]
```

DESCRIPTION

With no arguments, **getRecord** reads a registry record (an XML file) from standard input, and pretty prints the record to standard output, one field (name/value pair) per line. With an argument, **getRecord** outputs only the named field.

setRecord reads a registry record (an XML file) from standard input and writes a registry record (an XML file) to standard output. The command takes a set of command line arguments, each of the form **name=value**, where **name** identifies a field in the record and **value** corresponds to the value of that field. For each such argument, **setRecord** modifies the input record by setting the named field to the corresponding value. Note that the value may be given by a comma-separated list of values; e.g., **name=value1,value2,value3**. Also, since the value already present in the inputted record may be a set/list, the syntax **name+=value** and **name-=value** is used to append to and remove from that set (rather than replace the set with the given value). Finally, **setRecord** always outputs a valid registry record, even if the inputted record is empty. Thus, giving **setRecord** an empty file via standard input is a way to generate a default registry record suitable for further modification, although any unset values will be empty.

OPTIONS

-k, --keys display ssh keys (not shown by default)

NAME

getNodes, setNodes – get and set node fields in resource specification (rspec)

SYNOPSIS

```
getNodes [option]

setNodes [option] FILE
```

DESCRIPTION

getNodes reads an rspec (an XML file) from standard input, and writes the list of nodes contained in the rspec to standard output.

setNodes reads an rspec (an XML file) from standard input, modifies the list of nodes embedded in the rspec to correspond to those read from argument **FILE**, and writes the modified rspec (an XML file) to standard output. Argument **FILE** contains a list of nodes, one per line.

OPTIONS

-f ip, --format=ip	use IP addresses to identify nodes; by default, DNS names are used
-a NAME, --aggregate=NAME	perform get/set relative to named aggregate

For example

```
setNodes -a plc.eu ./nodelist < oldrspec.xml > newrspec.xml
```

adds the list of nodes contained in file **./nodelist** to the PlanetLab Europe aggregate in the inputted rspec, leaving the nodes specified for any other aggregates embedded in the rspec unchanged.