The Open Slice-based Facility Architecture (Open SFA)

Jordan Auge, Timur Friedman

The Open Slice-based Facility Architecture (Open SFA)

Jordan Augé & Timur Friedman

Revision History

Revision 0.1	2012-12-03	J

Table of Contents

About this document 1
Challenges in federating heterogeneous infrastructures 2
- Generalities 2
- SFA overview 2
- History 3
- SFA and the experiment lifecycle 3

SFA overview 3
- SFA entities 3
- The SFA registry and the record object type 5
- Slices and slivers 5
- The SFA manager interfaces and methods 5

Using SFA 6
- SFA for users 6
- SFA for testbed owners 6

Securely accessing entities in a distributed environment: naming, authentication & authorization 7
- Authentication 7
- Certificate, GID 7
- Authentication procedure 7
- Bootstrap procedure 7
- Authorization 8
- Generalities 8
- Credentials 8
- Rights and priviledges 8
- Credential delegation 9
- Credential verification 10
- Summary of a bootstrap process 10
- Hands-on illustration of the different concepts 10
- Tickets 12

SFA API reference 12
- Aggregate Manager API 12
- Registry API 17
- Common API 21

About this document

This document is intended as a step towards creating a globally-agreed upon standard for the Slice-based Facility Architecture, or SFA. At the time of writing, there exist some written SFA specifications, but as best we can tell they do not entirely correspond with written, currently functioning code. There also exist a number of functioning implementations, but these are not entirely documented.

We take as our starting point SFA as it is implemented by SFAWrap (http://sfawrap.info), which is a generic SFA wrapper for testbeds. SFAWrap is used by, among others, the PlanetLab, SensLAB, and FEDERICA testbeds, as well as by the FITeagle tool. As such , it is deployed both in the FIRE initiative in Europe and in the GENI initiative in the United States.

We dub this version the Open SFA specification, as we open it for discussion and contribution to stakeholders worldwide. This draft can be found on the OpenSFA website (http://opensfa.info), and those who wish to participate are invited to join the mailing list discuss@opensfa.info [mailto:discuss@opensfa.info].

References

- Generic SFA Wrapper implementation http://sfawrap.info
- Slice-base Federation Architecture v2.0 http://groups.geni.net/geni/wiki/SliceFedArch
- ProtoGENI API http://www.protogeni.net/trac/protogeni/wiki/API

Challenges in federating heterogeneous infrastructures

Generalities

Note

This section will describe the context in which SFA operates and the reasons why such a protocol was needed. It will also highlight the major requirements that influenced its design.

SFA overview

Note

present SFA, refer to original documents, timeline, planetlab

SFA has been designed to provide a minimal set of functionalities, a *thin waist* if you will, that a testbed can implement in order to enter into a global and interoperable federation. An experimenter in an SFA-based environment can transparently browse resources on any federated testbed, and allocate and reserve those resources.

Because of the potential for a very large number of testbeds, a global federation architecture faces a serious scalability issue. SFA introduces a fully distributed solution in which each peer testbed serves as the authority of reference for the resources that it brings, and each user community, along with its experiments, is represented by an authority (possibly, but not necessarily identified with an individual testbed).

Under the SFA architecture, there is a separation between what is generic and what is testbed-specific. Testbed-specific information is captured in a resource model, called a resource specification (RSpec), which is an XML transported by the SFA layer. SFA itself does not cover such aspects as resource model, policies, reservations or measurements. These functionalities should instead be implemented on top of SFA.

History

SFA was conceived of by Larry Peterson of Princeton University in the context of efforts to create a global PlanetLab federation. The first federation of computer networking testbeds was set up between Princeton's PlanetLab Central and UPMC's and INRIA's PlanetLab Europe, starting in 2006, as part of the EU's OneLab project. This initial federation was based on the pragmatic solution of synchronizing the central databases of each of the federated entities. In this way, users of each testbed gained full access to the resources of the other. As this solution worked in the context of two peers, but was clearly not scalable, SFA pointed the way forward. The first working deployment of SFA code was developed jointly by Princeton and INRIA, the latter working in the context of the EU's OneLab2 project. Starting in 2008, SFA was used to extend PlanetLab federation to other peers, such as PlanetLab Japan and EmanicsLab. Simultaneously, SFA was adopted as a control plane architecture by GENI, in which context written specifications ??? were drafted. Our description of SFA draws upon both the working code (from PlanetLab and from other, more recent, SFA implementations) and the written specification. These differ somewhat in their details, but agree on most of the main aspects.

SFA and the experiment lifecycle

SFA forms the control plane for browsing and reserving resources offers by a federation of networking testbeds. We often refer to the workflow followed by experimenters by *experiment lifecycle*. One common way to represent it is through the following steps: 1. Account management and authentication 2. Resource discovery 3. Resource reservation 4. Experiment configuration 5. Experiment running 6. Data collection 7. Resource release

Steps 1-3 and 7 are refer to as the *control plane*, while steps 4-6 correpond to the *experimental plane*. SFA only addresses the former. It is structed around four major components: * an '*API*': SFA's most visible part is an API allowing an interaction with the testbeds. It's purpose is to bring the minimal set of functionality allowing for a global federation. Without entering too much into details, we can identify three core SFA API calls that are in phase with the experimental lifecycle (+ those related to authentication and authorization, which are used out of band). These are: * for *Resource discovery*: 'ListResources` * for *Resource reservation*: `CreateSliver` * for *Resource release*: `DeleteSliver` We can also refer to another method that allows the discovery of capabilities and peers (`GetVersion`). A complete reference of the API is presented in Section ???. * an '*object management*' layer: SFA proposes a set of data types allowing the representation and identification of the different entities involved in the federation, and their associated data models. The next section (Section ???) introduces those concepts. * '*authentication and authorization*': A large part of the SFA specification proposes authentication and authorization mechanisms enabling a federation in a secure and distributed way, and supporting API calls between well-identified entities. They are described in detail in Section ???. * '*resource model*': Necessarily, the SFA layer cannot take into account the detailed aspects of each testbeds. Such specificities are abstracted into a *resource specification* document (RSpecs) that is transported on top of the SFA layer, while the latter only deals with high-level considerations that are common to all platforms. The last section (Section ???) is dedicated to RSpecs.

SFA overview

SFA entities

Object types. SFA designates a set of four main object types that represent the different entities involved in the testbed federation:

Authorities	These represent testbeds, parts of testbeds to which trust or rights may be delegated, and/or communities of users.

Resources	These consist of nodes, links, or any other experimental resource provided by the testbeds, and exposed to the users.
Users	These are experimenters wanting access to resources.
Slices	A slice is the basic unit of interaction between users and resources. One can think of a slice as corresponding to an experiment, and englobing all of the users and resources associated with that experiment. As its name suggests, slices play a central role in the SFA. They are presented in more details in the next section.

A hierarchical structure. In a centralized environment, it would be relatively easy to designate a central authority that would keep track of all the entities. This is less obvious in our decentralized context. The solution adopted by SFA is a classic one (the web in one example): we assume there will be a limited number of top-level domains. Because there will potentially be so many entities, each top-level domain is not required to know all of the entities under its responsibility. Rather, it enables a hierarchy, and each level in the hierarchy is only required to know about the level below.

SFA objects exist in a shared namespace, organized according to a hierarchy of authorities and sub-authorities. Trust relationships are based upon this hierarchy, with authorities vouching for the objects further down in the hierarchy. Figure ??? represents a part of the object hierarchy in the PlanetLab federation. We see the PlanetLab Europe (ple) and PlanetLab Central (plc) root authorities. We also see the sub-authorities, such as upmc and inria, princeton and columbia, that represent PlanetLab sites, which are the different universities and research labs that contribute resources to the testbed. These sub-authorities delegate the management of their own users to the root authorities. The figure highlights in red a user, represented by the HRN \texttt{ple.upmc.userA}, that corresponds to a user who has been vetted by the UPMC site under the PlanetLab Europe root authority. A registry is responsible for managing all of the objects under its branch, and will route other requests to the corresponding responsible authority.

Note

The need to distinguish hierarchy and routing

Naming. Objects of a given type are identified by a unique string composed of the chain of their parent authorities, dot-separated, followed by the name of the object within its authority. This is denoted as the Human-Readable Name of the object. An example of HRN is `auth.subauth.object`, which reflect the chain of parent authorities of the object. Note that this string can both designate a user and a slice.

In order to disambigate those cases, a Unique Resource Name (URN) is proposed, though less convenient for human users.

Note

we should explain the format of URNs here.

Finally, the term of XRN can either refer to a HRN or a URN. It is commonly used in the SFA API for functions that can work with either a HRN or a URN. When needed, a type parameter is also expected.

Identifiers. Objects are uniquely identified by a certificate, signed recursively by their home authority. We have thus a chain of certificates identifying all objects belonging to a top-level authority. The set of an object certificate and its parents is denoted GID. (Awkwardly, for a global federation scheme that is based on the notion that there is no one central authority, the term GID is said to stand for \emph{GENI identifier}.) We will see in Section ??? how these GIDs for the base for a secure distributed authentication and authorization system.

Note

certificates and GID. Refer to SFA2.0

The SFA registry and the record object type

A registry corresponds to a repository where the different objects and their properties are stored, in a distributed way following the naming hierarchy. More specifically, a registry associates an object XRN with its corresponding GID and other additional information.

Record data type

The entry characterizing an object in the registry is named a record, and has the following structure: * *'HRN'* * *'GID'* * *'type'*: (slice|node|user|authority|authority+sa|authority+am|authority+sm) * *'Info'* * Info = (PI[ ], Organization), if Type = SA * Info = (Owner[ ], Operator[ ], Organization), if Type = MA * Info = (URI, LatLong, IP, DNS), if Type = Component * Info = (URI, Researcher[ ], InitScript), if Type = Slice * Info = (PostalAddr, Phone, Email, AuthTokens[ ]), if Type = User

Slices and slivers

The notions of slice and sliver

Examples of slicing. Wired nodes = a virtual machine Wireless nodes = a node + a channel Slicing in time = reservation * 1 / N / infinite slices at the same time on a given resource

The SFA manager interfaces and methods

SFA defines a minimal set of API calls to enable interaction between the different actors of the federation, and that are implemented around three main components:

'Registry manager ®': This exposes objects that are managed by the federation. *'Aggregate manager (AM)'*: This exposes the resources of an individual testbed, or more generally, the resources that fall under a single authority. *'Slice manager (SM)'*: This exposes the resources from multiple, federated authorities and is used to track slice objects.

The API calls can be organized into three main categories:

'Object management': These calls manipulate registry objects through the classical list, create, read, update and delete functions. *Resource browsing and slice management'*: These calls associate resources to slices, as well as starting, stopping or getting the status of slices. *'Federation discovery'*: There is an API call that is used to obtain detailed information about the different federation services that are running, and to recursively discover peer platforms.

SFA is based on a web services API. To issue a call, a user must connect to a manager's XML-RPC interface via HTTPS, using their private key as a cypher, and passing as a first parameter the credential that shows that they are authorized to perform the operation.

Note

.API = language independent

XMLRPC alternatives. In our opinion, the choice of XMLRPC is an implementation choice. Other proposals such as REST ??? exists, and this is for example the choice made by the OMF-FA ??? implementation.

Illustration. Full API in a separate part also. He we should only explain the basics and give a few examples.

Registry Manager API

The *registry manager* is a component dedicated to the manipulation of this registry: creating, displaying, updating, deleting and listing objects.

Action on these objects through APIs. Secure invocation mechanisms: part of a separate part. Architecture could in principle support multiple such schemes.

Note

We can explain here that the registry is the perfect place for user management. NOTE: Relationship between registry and User/Slice/… authorities… NOTE: Explain that it issues credentials, refer to auth/auth section.

Note

How is the hierarchy decided ? NOTE: People might be confused that we have XXX.YYY.their_testbed NOTE: GENI clearing house = want to minimize the number of trusted entities. Is it because we have no certificate distribution system ?

1. cf get_trusted_certs ??

Aggregate and Slice Manager API

Using SFA

SFA for users

SFA clients

- sfi.py
- sface
- Omni
- Flack
- MySlice

SFA for testbed owners

SFA implementations

http://groups.geni.net/geni/wiki/GeniApi

Joining the SFA federation

In order for a testbed to become part of the current global federation enabled through SFA, given that a trust relationship has been established with at least one current member of the federation, there are two important

technical requirements that need to be fulfilled. First, the local testbed resources must be described in an RSpec that the testbed's aggregate manager can both send and understand. Second, a friendly user interface must be available for researchers to be able to browse the available resources, express their requirements, and reserve the desired set of resources of this testbed in a fashion that is consistent with the rest of the federation. The following two sections describe two software components, the Generic SFA Wrapper and MySlice, that propose to make this process easy for testbed owners and developers.

Securely accessing entities in a distributed environment: naming, authentication & authorization

Authentication and authorization are two essential notions that are too often mixed up. They respectively answer the two following questions: *Who am I ?* and *What can I do?*.

SFA defines a set of functionalities to manage the different entities involves in a federation. The remainder of this section describes these functionalities, as well as how they are abstracted into various managers.

Authentication

Certificate, GID

SFA currently bases its authentication mechanism on a public key infrastructure, where each object has a keypair (a public and a private key) and is associated with a signed certificate, called a GID, that is stored in a registry. The certificate is used for authentication, following the same principles as user and website authentication on the web. It is a X.509 certification ??? that associates the object's HRN with its public key, and that is signed recursively by each parent authorities up to the root.

Authentication procedure

Let's consider a user U willing to authenticate to a server S thanks to its GID. U needs to establish a SSL connection (in our case a XMLRPC connection over HTTPS) cyphered with the private key associated to the GID.

If the SSL session succeeds, S only has to validate the GID to be able to recognize the user identified by the related HRN. The only requirement for this is that S trusts the root certificate that has signed the GID. In no way either U or S have to contact the user's home authority that has established the GID; the possession of the GID by the user if sufficient and can be done out-of-band through a bootstrap procedure. If U's root authority is trusted by S, the chain of signatures proves that the owner of this keypair is really U.

This process which is natural for users can also be used by authorities, for peer communications, but also for resources for example (even though it is not commonly used today, it might be of interest for a testbed building only on the SFA).

Bootstrap procedure

Initially, a user U is only supposed to posses a keypair. A bootstrap procedure (such as user registration) is necessary to that its home authority H knowns its public key, and is able to associate it to its HRN.

From its keypair, U can generate a self-signed certificate it will use for the SSL connection. This is sufficient for H to authenticate the user since it knows his user account, which was not the case in the

previous paragraph. This allows the user to retrieve its GID by contacting the registry of its home authority (`GetGid` method). It can then use it to contact all other federated entities.

Authorization

Generalities

In SFA, at present, an entity can be authenticated even if later it turns out that it is not allowed to perform actions or access resources. This second step is called authorization, and it depends on the local policies in use on the various testbeds. As for authentication, there are many possible ways to perform authorization in a distributed environment. For instance, A, a user of testbed B, might also want to get authenticated and authorized by testbed C, which might never have heard of A or B, but trusts them indirectly because C trusts their root authority.

Credentials

Currently, SFA implements the notion of a credential, which is a signed XML document that proves that an entity has a set of rights relating to another one, and states whether or not its has the possibility to delegate those rights. Such credentials can be used to establish the various trust relationships necessary to run a federated platform. For example, a slice credential might allow a privileged user to create a slice, while a less privileged user might only be allowed to perform operations on an existing slice.

More precisely a credential stores the following information: '*caller*': identified entity to which the credential has been issued, characterized by its HRN and GID. Most of the time, the caller is a user (or an authority); '*object*': identifies the object for which the credential holds. The type of the object determines the type of credential: user credential, slice credential or authority credential; '*expires*': a credential is issued for a limited lifetime; '*priviledges*': a set of priviledges that are assigned to the caller with respect to the object, '*delegate*': each priviledge is annotated with a flag indicating whether it can be further delegated.

There is currently a debate in the SFA community as to whether to move to an attribute-based access control (ABAC) authorization mechanism, in which a user could assemble a set of signed clauses from various entities, and use them to construct a proof that they (the user) indeed have the rights that they claims. Shibboleth ???, which is used to managed a federation of identity providers for national research networks (NRENs), is also a candidate for a future authorization system.

Note

Legacy credentials used to be an equivalent tuple stored in the *subjectAltName* of an X509 certificate. This behaviour is deprecated and not the subject of the present document.

Rights and priviledges

Table 1. Rights

Right	Priviledges
user	refresh, resolve, info
sa	authority, sa
ma	authority, ma
authority	authority, sa, ma
slice	refresh, embed, bind, control, info

Right	Priviledges
component	operator

Note

Rights seem to be somewhat redundant with the type of the credential. For example, a "sa" credential implies the authority right, because a "sa" credential cannot be issued to a user who is not an owner of the authority.

Table 2. Priviledges

Priviledge	Operations
authority	register, remove, update, resolve, list, getcredential, *
refresh	remove, update
resolve	resolve, list, getcredential
sa	getticket, redeemslice, redeemticket, createslice, createsliver, deleteslice, deletesliver, updateslice, getsliceresources, getticket, loanresources, stopslice, startslice, renewsliver, deleteslice, deletesliver, resetslice, listslices, listnodes, getpolicy, sliverstatus
embed	getticket, redeemslice, redeemticket, createslice, createsliver, renewsliver, deleteslice, deletesliver, updateslice, sliverstatus, getsliceresources, shutdown
bind	getticket, loanresources, redeemticket
control	updateslice, createslice, createsliver, renewsliver, sliverstatus, stopslice, startslice, deleteslice, deletesliver, resetslice, getsliceresources, getgids
info	listslices, listnodes, getpolicy
ma	setbootstate, getbootstate, reboot, getgids, gettrustedcerts
operator	gettrustedcerts, getgids
*	createsliver, deletesliver, sliverstatus, renewsliver, shutdown

Note

"*" is a privilege granted by ProtoGENI slice authorities, and we give it access to the GENI AM calls

Credential delegation

Because authentication requires the possession of a private key, a delegation mechanism has been implemented in SFA so that a user could perform actions for which it has been delegated the rights, on behalf of another user. A delegated credential has the same structure which states that the delegee has some rights on an entity and is signed by the delegating user. It also encloses the original credential(s), proving that the original user has both the delegated privileges and the right to delegate them.

Credential verification

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.

Verify that: * All of the signatures are valid and that the issuers trace back to trusted roots (performed by xmlsec1) * The XML matches the credential schema * That the issuer of the credential is the authority in the target's urn * In the case of a delegated credential, this must be true of the root * That all of the gids presented in the credential are valid * Including verifying GID chains, and includ the issuer * The credential is not expired

For Delegates (credentials with parents) * The privileges must be a subset of the parent credentials * The privileges must have "can_delegate" set for each delegated privilege * The target gid must be the same between child and parents * The expiry time on the child must be no later than the parent * The signer of the child must be the owner of the parent

Verify does **NOT** * ensure that an xmlrpc client's gid matches a credential gid, that must be done elsewhere

Verify issuer: Make sure the credential's target gid (a) was signed by or (b) is the same as the entity that signed the original credential, or (c) is an authority over the target's namespace. Also ensure that the credential issuer / signer itself has a valid GID signature chain (signed by an authority with namespace rights).

Summary of a bootstrap process

Hands-on illustration of the different concepts

It is important to note that all mechanisms used by SFA rely on standard mechanisms and tools. In this section, we propose a look at how these objects can be manipulated though standard tools, namely `openssl` for certificates, and `xmlsec1` for credentials. This is of course fully handled by the existing SFA clients. In this section, and we use the commandline tool `sfi.py` originating from the Generic SFA Wrapper implementation [1].

We take the example of a user '*myuser*' accessing the testbed '*mytestbed*'.

SSH keypair creation.

```
ssh-keygen -t rsa
cp ~/.ssh/id_rsa.pub myuser.key
```

Generation of the self-signed certificate.

```
# Certificate Signing Request (CSR)
openssl req -new -key myuser.pkey -out myuser.csr

# Certificate
openssl x509 -req -days 365 -in myuser.csr -signkey myuser.pkey -out myuser.cert
(365x5 = ???, give only CN)

# Display the generated certificate
openssl x509 -text -in myuser.cert -noout
```

[1] http://www.sfawrap.info

GID repatriation.

```
# Retrieving the GID through a SFA call
sfi.py gid mytestbed.myuser

# Display the GID
openssl x509 -text -in sfi_files/mytestbed.myuser.gid
openssl x509 -text -in sfi_files/mytestbed.myuser.user.gid

# Need to explain the difference between both GIDs...

# Displaying the hierarchy of signatures in the GID
perl -n0777e 'map { print "---\n"; open(CMD, "| openssl x509 -noout -subject -issu

# Verifying the GID
openssl verify -CAfile trusted_roots/mytestbed.gid sfi_files/myteutotstbed.myuser.
openssl verify -CAfile trusted_roots/mytestbed.gid sfi_files/mytestbed.myuser.user

# Generating a PKCS12 token for web authentication
openssl pkcs12 -export -inkey myuser.pkey -in sfi_files/mytestbed.myuser.gid -out
```

Use of the `sfi.py` client. The first time a command is issued through `sfi.py`, the bootstrap process we have illustrated is performed automatically, and the different objects retrieved are cached in the `~/.sfi` directory.

```
sfi.py list mytestbed

# The following files have been generated in ~/.sfi:
#  - mytestbed.myuser.pkey        (user private key)
#  - mytestbed.myuser.sscert      (self-signed certificate)
#  - mytestbed.myuser.user.gid    (user GID)
#  - mytestbed.myuser.user.cred   (user credential)

# Display the generated self-signed certificate
openssl x509 -text -in ~/.sfi/mytestbed.myuser.sscert
```

Credential repatriation. Just like the GID, credentials will be automatically repatriated by `sfi.py` when needed.

```
# The user credential is gathered through the bootstrap process we already
performed, and should be available in ~/.sfi/mytestbed.myuser.user.cred.

# List the resources associated to a slice
sfi.py resources mytestbed.slice1

# We should now also have a slice credential in ~/.sfi/mytestbed.slice1.slice.cred
```

Credentials can be inspected thanks to a regular text editor.

In order to verify the XML signature, the `xmlsec1` tool can be used:

```
xmlsec1 --verify --node-id "ref0" --trusted-pem trusted_roots/mytestbed.gid sfi_fi

# In case of a delegated credential, the parent credential has the node-id
# 'ref0', and the newly delegated credential has the id 'ref1'
```

```
xmlsec1 --verify --node-id "ref1" --trusted-pem trusted_roots/mytestbed.gid sfi_fi
```

Tickets

TODO

SFA API reference

The different methods in this section are presented by alphabetical order, grouped by component.

Note

Describe the kind of information that is given for each method.

Aggregate Manager API

Method `CreateSliver`

Description. Allocate resources to a slice. This operation is expected to start the allocated resources asynchornously after the operation has successfully completed. Callers can check on the status of the resources using SliverStatus.

Prototype. `CreateSliver(slice_xrn, creds, rspec, users, options)`

Parameters

`slice_xrn` XRN of the slice to allocate to (string)

`creds` Credential string or list of credential strings

`rspec` RSpec to allocate (string)

`users` A list of user information under the form:

- '*urn*': URN of the users that is allowed to access the slice
- '*keys*': List of ssh (RSA) keys in a string format

`options` Dictionary of options

- '*call_id*':
- self.ois ?

Return value. The return value is an ReturnValue associative array with the following fields: * '*status*': either *success* or *exception* * '*aggregate*': * '*elapsed*': duration of the API call * '*rspec*': the manifest RSpec or the created slice/sliver (only if status == *success*) * '*exc_info*': exception information (only if status == *exception*)

Interfaces. AM, SM

Notes. This method will eventually trigger the creation of the slice and related users on a third-party testbed. This explains why minimal users information need to be passed as parameters when calling this

function (cannot be NULL). Such information can be retrieved thanks to the `Resolve` method. This also explains why this methods does not apply to CM while `DeleteSliver` does.

Anything not contained in this request will be removed from the slice.

The RSpec must explicitely allocate slivers (presence of <sliver_type> or <sliver> elements).

The SM loops through all aggregates and combines the received RSpecs. The returned RSpecs by the SM has a statistics section. The SM drops eventually received statistics sections.

The slice is automatically started (might be dependent on the driver)

Note

sfatables ?

Note

What about various Rspecs formats ? What about unknown formats ? NOTE: what about slice tags ? NOTE: What is sliver_type ?

Example. The associated `sfi.py` command is `create_sliver`

Method `DeleteSliver`

Description. Remove the slice from all nodes and free the allocated resources

Prototype. `DeleteSliver(xrn, creds, options)`

Parameters

'*xrn*'	XRN of the slice to instanciate
'*creds*'	(List of) credential(s) string(s) specifying the rights of the caller
'*options*'	Associative array of options with the following fields • '*call_id*' (optional)

Return value. 1 if successful, faults otherwise

Interfaces. CM, AM, SM

Method `GetTicket`

Description. Retrieve a ticket for the specified slice. The ticket is filled in with information from the testbed. It includes resources, and attributes such as user keys and initscripts.

Prototype. GetTicket(xrn, creds, rspec, users, options)

Parameters. xrn name of the slice to retrieve a ticket for (hrn or urn) cred credential string or List of credential rspec resource specification dictionary user List of user information options Options call_id (.ois())

Return value. String representation of the ticket.

Interfaces. AM, SM

Example. The corresponding operation in `sfi.py` is `get_ticket`.

Method `ListResources`

Description. Returns information about available resources or resources allocated to this slice

Prototype. ListResources(creds, options)

Parameters

`creds` (List of) credential string(s) specifying the rights of the caller.

`options` Associative array of options with fields:

- '*call_id*': always send call_id to v2 servers
- '*cached*': ask for cached values if available
- '*geni_slice_urn*'
- '*geni_rspec_version*': rspec return format. client must specify a version through geni_rspec_version or rspec_version. Example: `{type: geni, version: 3.0}`
- '*rspec_version*': alternative to geni_rspec_version
- '*geni_compressed*': zlib compressed data with base64 encoding
- '*info*':
- '*list_leases*': (leases|resources)

Interfaces. AM, SM

Method `ListSlices`

Description. List instanciated slices

Prototype. ListSlices(creds, options)

Parameters

`cred` credential string specifying the rights of the caller, or list of credentials

`options` Associative array of options with fields:

- '*call_id*'

Return value. A list of URNs

Interfaces. CM, AM, SM

Example. The corresponding `sfi.py` method is `slices`

Method `RedeemTicket`

Prototype. RedeemTicket(ticket, creds)

Parameters

`ticket` String representation of a sfa ticket

`creds` (List of) credential string(s) specifying the rights of the caller.

Return value. 1 if successful

Interfaces. CM

Method `RenewSliver`

Description. Renews the resources in a sliver, extending the lifetime of the slice.

Prototype. `RenewSliver(slice_xrn, creds, expiration_time, options)`

Parameters

`slice_xrn` XRN of the slice to renew.

`creds` (List of) credential string(s) specifying the rights of the caller.

`expiration_time` Requested expiration time in RFC 3339 format (string).

`options` Associative array of options with fields:

- *'call_id'*:
- *'(+ ois)'*

Return value. {"aggregates": results, "code": {"geni_code": geni_code}, "value": geni_value, "output": geni_output}

internally (interpretation might be wrong in other parts of this document) {"code": {"geni_code": 0}, value: result} {"aggregate": aggregate, "exc_info": traceback.format_exc(), "code": {"geni_code": -1}, "value": False, "output": ""}

bool success or failure

Interfaces. AM, SM

Example. The corresponding `sfi.py` command is `renew`.

Method `reset_slice`

Description. Reset the specified slice.

Prototype. reset_slice(cred, xrn, origin_hrn=NULL)

Parameters

`cred` (List of) credential string(s) specifying the rights of the caller.

`xrn` XRN of the slice to renew.

`origin_hrn` XRN of the original caller, or NULL

Return value. 1 is successful, faults otherwise

Interfaces. CM, AM, SM

Example. The corresponding `sfi.py` command is `reset`.

Method `Shutdown`

Description. Perform an emergency shut down of a sliver. This operation is intended for administrative use. The sliver is shut down but remains available for further forensics.

Prototype. `Shutdown(slice_xrn, cred)`

Parameters

`slice_xrn` XRN of the slice to shutdown

`cred` (List of) credential string(s) specifying the rights of the caller.

Return value. Bool success of failure

Notes. Shutdown calls the `Stop` method.

Interfaces. AM, SM

Method `SliverStatus`

Description. Get the status of a sliver

Prototype. `SliverStatus(slice_hrn, cred, options)`

Parameters

`slice_xrn` XRN of the slice.

`cred` (List of) credential string(s) specifying the rights of the caller.

`options` Associative array of options with fields:

- '*call_id*':

Return value. A ReturnValue dictionary / A list of associative arrays with fields: * '*geni_resources*' (for SM and AM) * '*geni_urn*' (for SM) * '*pl_login*' (for SM) * '*status*': (*ready*|*unknown*) (for SM) *unknown* is used if no '*geni_resources*'

Interfaces. CM, AM, SM

Example. The corresponding `sfi.py` command is `status`.

Method `Start`

Description. Start the specified slice.

Prototype. Start(xrn, creds)

Parameters

`xrn` XRN of the slice to start.

`cred` (List of) credential string(s) specifying the rights of the caller.

Return value. 1 is successful, faults otherwise

Method `Stop`

Description. Stop the specified slice.

Prototype. Stop(xrn, creds)

Parameters

`xrn` XRN of the slice to stop.

`cred` (List of) credential string(s) specifying the rights of the caller.

Return value. 1 is successful, faults otherwise

Method `UpdateSliver`

Description. Allocate resources to a slice. This operation is expected to start the allocated resources asynchornously after the operation has successfully completed. Callers can check on the status of the resources using `SliverStatus`.

Prototype. `UpdateSliver(slice_xrn, creds, rspec, users, options)`

Parameters

`slice_xrn` XRN of the slice to update.

`cred` (List of) credential string(s) specifying the rights of the caller.

`rspec` RSpec to allocate.

`users` List of user information.

`options` Associative array of options with fields:

Return value. allocated rspec

Interfaces. AM, SM

Note. Depending on the implementation, this method might be an alias for `CreateSliver`.

Registry API

Method `CreateGid`

Description. This method creates a signed certificate in the registry for the referenced object. In addition to being stored at the SFA level, a call to the testbed will also be performed to create the record at the testbed level.

Prototype. `CreateGid(creds, xrn, cert=NULL)`

Parameters

`cred` (List of) credential string(s) specifying the rights of the caller

`xrn` XRN of the certificate owners

`cert` caller's certificate (default: None)

Return value. The GID string representation

Example. The corresponding `sfi.py` command is `create_gid`.

Method `GetCredential`

Description. Retrieve a credential for an object.

Prototype. `GetCredential(cred, xrn, type)`

Parameters

`cred` (List of) credential string(s) specifying the rights of the caller.

`xrn` XRN of the object

`type` type of the object (user | slice | node | authority | NULL)

Return value. Credential string.

Notes. If the credential argument is NULL, then the behaviour of the method might revent to `GetSelfCredential`.

Method `GetGids`

Description. Get a list of record information (hrn, gid and type) for the specified hrns.

Parameters

`xrns` (List of) XRN(s)

`cred` (List of) credential string(s) specifying the rights of the caller.

Return value. An associative array with fields: * '*hrn*' * '*type*' * '*gid*' (issued from Resolve).

Notes. This methods is calling `Resolve`.

Example. This method has no entry point in `sfi.py`.

Note

Is this method important ? When is it called ?

Method `GetSelfCredential`

Description. Retrive a credential for an object thanks to self signed certificates as the SSL cert (see the bootstrap process / how to get credentials).

Prototype. `GetSelfCredential(cert, xrn, type)`

Parameters

`cert` certificate string

`xrn` human readable name of object (hrn or urn)

`type` type of object (user | slice | sa | ma | node)

Return value. Credential string

Notes. GetSelfCredential a degenerate version of GetCredential used by a client to get his initial credential when de doesnt have one. This is the same as GetCredential(…, cred = None, …)

The registry ensures that the client is the principal that is named by (type, name) by comparing the public key in the record's GID to the private key used to encrypt the client side of the HTTPS connection. Thus it is impossible for one principal to retrieve another principal's credential without having the appropriate private key.

Method `List`

Description. List the entries/records in a named authority registry.

Prototype. List(xrn, creds, options={})

Parameters

`hrn` human readable name of authority to list (hrn or urn)

`cred` (List of) credential string(s) specifying the rights of the caller

`options` associative array of options with fields:

- '*recursive*': (same as putting a * at the end of the hrn) consider subauthorities

Return value. A list of record dictionaries with fields: * '*hrn*': * '*type*': * '…': ?

Notes. Load all know registry names into a prefix tree and attempt to find the longest matching prefix * if there was no match then this record belongs to an unknow registry * if the best match (longest matching hrn) is not the local registry, forward the request

Example. The corresponding `sfi.py` command is `list`.

```
sfi.py list mytestbed
sfi.py list mytestbed.subauthority
```

Method `Register`

Description. Regiter a new object (record) within the registry. In addition to being stored at the SFA level, the appropriate records will also be create at the testbed level.

Prototype. Register(record, creds)

Parameters

`record_dict` Record dictionary containing record fields

`cred` credential string, or list of credentials

Return value. String representation of GID

Example. The corresponding `sfi.py` commmand is `add`.

Method `Remove`

Description. Remove the named object from the registry. If the object also represents a testbed object, the corresponding record will be also removed from the testbed.

Prototype. `Remove(xrn, creds, type)`

Parameters

`xrn` XRN of the record to remove

`creds` credential string, or list of credentials

`type` record type, or type not specified. The type can be *all*, *.

Return value. 1 if successful, faults otherwise

Example. The corresponding `sfi.py` commmand is `remove`.

Method `Resolve`

Description. Resolve, show details about the named registry record(s).

Prototype. `Resolve(xrns, creds)`

Parameters

`xrns` XRN(s) to resolve

`creds` (List of) credentials string(s) specifying the rights of the caller.

Return value. A list of record dictionaries or empty list

Example. The corresponding `sfi.py` commmand is `show`.

Method `Update`

Description. Update a object (record) in the registry. This might also update the tested information associated with the record. Depending on the implementation, the SFA fields (name, type, GID) might be fixed.

Prototype. `Update(record_dict, creds)`

Parameters

`record_` An associative array representing the record to be updated.

`type` Type of the record (mandatory)

`cred` (List of) credentials string(s) specifying the rights of the caller.

Return value. 1 if successful, faults otherwise

Method ``

Common API

Method `GetVersion`

Description. Returns a SFA server version information.

Prototype. GetVersion(options={})

Parameters

`options` associative array of options

Return value. ReturnValue with the version dict as a value

- '*code_tag*': version_tag, (eg. like 2.1-11) (for RM, CM, AM, SM)
- '*code_url*': Source code management (SCM) URL (for RM, CM, AM, SM)
- '*hostname*': hostname of the component (as retrieved by gethostname) (for RM, CM, AM, SM)
- '*interface*': name of the SFA component (registry|aggregate|slicemgr|component) (for RM, CM, AM, SM)
- '*sfa*': SFA version (currently: 2) (for RM, AM, SM)
- '*geni_api*': GENI API version (currently: 2) (for RM, AM, SM)
- '*geni_api_versions*': {*2*: *http://HOST:PORT*} (for AM, SM)
- '*hrn*': HRN (for RM, AM, SM)
- '*urn*': URN (for RM, AM, SM)
- '*peers*': list of peers (for RM, SM)
- '*geni_request_rspec_versions*': request_rspec_versions (for SM)
- '*geni_ad_rspec_versions*': ad_rspec_versions (for SM)
- '*testbed*': [*myplc*] (for CM)

Interfaces. R, CM, AM, SM

Method `get_trusted_certs`

Description. ??

Prototype. get_trusted_certs(cred = None)

Parameters

`cred` (List of) credential string(s) specifying the rights of the caller, or None.

Return value. list of gid strings

Notes. If cred is not specified just return the gid for this interface. This is true when when a peer is attempting to initiate federation with this interface

Interfaces. R, SM, AM