# Table of Contents

**The DETER System Programming Interface**

- Introduction 4
- The Fundamental Abstractions 4
- The Implementation 5
  - DETER Abstractions and the Existing Codebase 5
  - Delivering the SPI Through SOAP 7

**Understanding The SPI**

- Services 7
- Profiles 7
- Users 9
  - Authentication 9
  - Profile Management 10
  - Notifications 10
  - Password Management 11
  - Creating New Users 11
  - Administrative Operations 11
- Projects 11
  - Project Permissions 12
  - Creating A Project 12
  - Adding Users To A Project 13
  - Project Ownership 13
  - Changing Project Permissions 13
  - Project Profile Management 13
  - Removing a Project 13
  - Viewing Projects 13
  - Administrative Operations 14
- Circles 14
  - Circle Names 14
  - Circle Permissions 14
  - Creating A Circle 15
  - Adding Users To A Circle 15
  - Circle Ownership 15
  - Changing Circle Permissions 15
  - Circle Profile Management 15
  - Removing a Circle 15
  - Viewing Circles 16
  - Administrative Operations 16
- Experiments 16
  - Aspects 16
    - Layout Aspect 17
  - Experiment Names 17
  - Experiment Permissions 17
  - Creating A New Experiment 17
  - Adding and Changing Experiment Aspects 17
  - Adding and Changing Experiment Access Control Lists 18
  - Experiment Ownership 18
  - Experiment Profile Management 18
  - Removing an Experiment 18
  - Viewing Experiments 18
  - Administrative Operations 19
- Libraries 19
  - Library Names 19
  - Library Permissions 19
  - Creating A Library 20
  - Adding Experiments To A Library and Removing Them 20
  - Adding and Changing Library Access Control Lists 20
  - Library Ownership 20
  - Library Profile Management 20
  - Removing a Library 20
The DETER System Programming Interface

Introduction

The DETER System Programming Interface (SPI) is a standard interface for manipulating fundamental DETER abstractions used to manage testbed resources and to marshall those resources into network and cybersecurity experiments. The SPI is analogous to an operating system’s system call interface. It provides the fundamental abstractions that enable application programmers to make tools that are useful to researchers.

There are two motivations for building the SPI: first that the SPI is a context and programming framework for new DETER abstractions that subsumes the existing Emulab codebase cleanly. Second the SPI provides a context in which new developers can extend those abstractions to provide new functionality and to improve their scaling properties.

The SPI is fundamentally a tool-builder’s interface. Researchers performing experiments or students performing class exercises are unlikely to access SPI methods directly, though we expect that higher level tools and interfaces will make the fundamental abstractions visible.

The Fundamental Abstractions

There are 6 first class abstractions that make up the DETER SPI. These are:

Users
A researcher who builds and carries out experiments using resources controlled by the testbed.

Projects
Administrative groupings of users that embody the trust chains that underly researcher access. A researcher has access to the testbed as a whole because they are a member of an approved project. Trusted users can add other users to an approved project, reducing the administrative overhead. Because creating a project directly involves committing trust to a user to both use testbed resources and allow others to do so, a project is more heavyweight than a circle.

Circles
Groupings of trusted users that embody the sharing of data and resources among researchers. Adding or removing users from circles gives them permission to access resources in use for experimentation or excludes them from such resources. Circles are more dynamic than projects, and represent collaboration and sharing among users who are already trusted by the testbed administration.

Experiments
A description of a research activity. An experiment is composed of aspects, each of which embodies some facet of the activity. Currently the SPI recognizes 4 aspects:

- **Layout**: the arrangement and specification of an experiment’s elements that make up the experimental apparatus
- **Procedure**: The steps taken to carry out an experiment
- **Data Management**: The resources and processes used to collect and process data generated by the experiment
- **Constraints**: Correctness requirements that must be met in a valid experiment.

Additional aspects can be specified by users and accessed by running experiments. We document a plug-in interface (below) that explains how new aspects can enter the system. Access to experiments and the resources attached to them is controlled by assigning rights to circles.

Libraries
Collections of experiments grouped by function or semantics. Libraries allow researchers to organize their experiments and share them with other researchers (by assigning rights to circles).

Realizations
A realization is the binding of resources to an experiment. Users can inspect and modify such bindings in carrying out their experiments.

Projects and circles are related in that they control access to testbed resources.

Membership in an approved project means a user can access testbed resources in general. They’ve been trusted directly or by transitivity by the testbed administration to have "an account" to which usage can be charged and responsibility delegated. Projects must be approved by administrators to confer rights.

Membership in a circle allows users to specify fine grained access between one another. None of the circle memberships are effective unless the user is also in an approved project. Circles do not need to be vetted by admins because their powers are gated by projects.

Being in a project means a user can use DETERLab. Being in the circle of users that can read an experiment means a user can use that particular experiment in that particular way (assuming that the user is in an approved project).

We describe each of these abstractions in more detail below. They are interrelated and the descriptions link to one another in this document. We intend for readers to jump between sections to help intuition.
The Implementation

This section motivates and explains two of the implementation design decisions that underlie the SPI: expanding the implementation beyond the Emulab abstractions and implementation, and using SOAP and web services to deliver the service.

DETER Abstractions and the Existing Codebase

The DETER research agenda drives the new abstractions, and it differs somewhat from the abstractions that drive the Emulab-derived code base. DETER is interested in sharing experiment infrastructure - including conducting experiments where multiple parties manipulate the environment based on different views of the world and different goals (multi-party experiments). Also, DETER is interested in multi-fidelity, large-scale experiments where the representation of the various parts of the experiment is guided by the researcher's design of the experiment. Finally, we think of experiments as having aspects that go well beyond the definition of the topology or layout, and the abstractions need to support this.

The Emulab code is based on simpler topology and experimentation models, and while its developers have been able to extend it to do more, our changes take the abstractions far enough that relegating the Emulab code to allocating local resources seems the cleanest way forward. Functions including user and project management will move into the SPI, though subsets of that function (basically current Emulab functionality) will remain available through the legacy interface.
As a last practical point Emulab-based codebase blurs the lines of some implementations between the webserver interface and the backend code. Building the SPI offers DETER a chance to solidify that interface and enable developer to build multiple applications on the DETER abstractions.
Delivering the SPI Through SOAP

The SPI is built on the SOAP web services platform as a compromise between accessibility, extensibility, and support from tools. An important goal of the SPI is to allow multiple applications to use the DETER testbed. A web front-end for teaching cybersecurity classes that coexists with a web front-end for experimenting on Internet anonymity are such a pair of applications. That may even look like two different testbeds from the outside.

To support that, the SPI must be platform independent. A web browser is a powerful application platform that the SPI must support. The SOAP/WSDL standards are a proven technology for such access. SOAP may be somewhat stodgier than a JSON/REST interface, but the differences can be constrained to encodings, and many tools for generating SOAP interfaces provide migration paths to other web services platforms.

Presently we take advantage of the fairly rich SOAP toolspace and have been able to use SOAP for access from a range of platforms from web browsers to multiple standalone programming languages.

The rest of this document goes into more detail on how to use the SPI.

Understanding The SPI

This section describes the SPI abstractions and operations that application writers will use. We are concerned with how to put the concepts into use and how they interrelate with each other. Below we describe the detailed interface.

Services

The SPI is broken up into services that implement the various abstractions. Each major abstraction presents its own service, and there are a few others for administration and information. Each service is a separate SOAP/WSDL service and has its own contact point. We break them up this way to keep the specification reasonably sized for both people and computers.

The services in the SPI are:

- ApiInfo
  - Version information and client and server certificate distribution.
- Admin
  - Low level testbed administration
- Circles
  - Manipulation of Circles
- Experiments
  - Manipulation of Experiments
- Libraries
  - Manipulation of Libraries
- Projects
  - Manipulation of Projects
- Users
  - Manipulation of Users

Profiles

A profile is a collection of metadata attached to an abstraction instance, primarily to help users and administrators understand what the instance is being used for. Users, projects, circles, experiments, and libraries all have profiles attached to them which describe those things. For example, a user profile includes the researcher’s name and an e-mail address.

Profiles are self-describing, so that applications have guidance is what is being presented and how to display it. Each of the abstractions above has an operation called getProfileDescription in its associated service that returns an array of all the valid fields - called attributes - in that abstraction’s profile. Each element of that array - each attribute - includes additional metadata that describes its purpose and presentation:
Each instance has an associated operation, `getUserProfile` (projects have `getProjectProfile`, experiments have `getExperimentProfile`) that returns the profile for that instance as an array of attributes, as above. This profile includes the values for all the (readable) attributes.

Each instance supports a `changeUserAttribute` operation that allows the application to change the instance's attribute values. `changeUserAttribute` is used to change a user's phone number. Optional attributes can be deleted through this interface.

An application will use the profile operations in the following ways:

- When creating an instance that requires a profile, the application will request the profile description using `getProfileDescription`, fill in non-optional fields (probably based on user input) and include that profile data with the creation request. All abstractions that have profiles attached require a valid profile as a parameter to the operation that creates them.
- When displaying an instance that supports a profile, the application may request the profiles for various instances using `getUserProfile` et al., and display all or some of the profile data.
- Applications that manage abstractions with profiles may allow users to change the contents of the profile using `changeUserAttribute`, et al.

There is an administrative interface for changing the attributes in each abstraction's profile, but we expect the profile contents to remain relatively constant over time.

Currently the attributes that DETERLab includes in abstraction profiles are as follows:

For users:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Optional</th>
<th>Access</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name</td>
<td>false</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>title</td>
<td>Title</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>email</td>
<td>E-mail</td>
<td>false</td>
<td>READ_ONLY</td>
<td>STRING</td>
</tr>
<tr>
<td>affiliation</td>
<td>Affiliation</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>affiliation_abbrev</td>
<td>Affiliation (abbreviated)</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>URL</td>
<td>URL</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>address1</td>
<td>Address</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>address2</td>
<td>Address Line 2</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>city</td>
<td>City</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>state</td>
<td>State</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>zip</td>
<td>Postal Code</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>country</td>
<td>Country</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>phone</td>
<td>Phone</td>
<td>false</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
</tbody>
</table>

All users are required to have a name, e-mail address, and a phone number present in their profile, but may only change the name and phone number. (The e-mail address is used to validate the accounts.)

The e-mail and phone number attributes have formats attached:

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Format Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>email</td>
<td>[^\s@]+@[^\s@]+</td>
<td>A valid e-mail address</td>
</tr>
<tr>
<td>phone</td>
<td>[0-9-\s.,()-+]+</td>
<td>Numbers, whitespace, parens, plus signs, and dots or dashes</td>
</tr>
</tbody>
</table>

Because we expect applications to edit user profiles, there is layout information as well:
### For Projects:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Optional</th>
<th>Access</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>Description</td>
<td>false</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>funders</td>
<td>Funders</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>affiliation</td>
<td>Affiliation</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>URL</td>
<td>URL</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
</tbody>
</table>

### For Circles:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Optional</th>
<th>Access</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>Description</td>
<td>false</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
<tr>
<td>email</td>
<td>Email</td>
<td>true</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
</tbody>
</table>

### For Experiments:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Optional</th>
<th>Access</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>Description</td>
<td>false</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
</tbody>
</table>

### For Libraries:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Optional</th>
<th>Access</th>
<th>DataType</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>Description</td>
<td>false</td>
<td>READ_WRITE</td>
<td>STRING</td>
</tr>
</tbody>
</table>

## Users

A user represents a researcher using the testbed. Most operations in the SPI are carried out on behalf of a user. Users must authenticate themselves to the testbed in order to carry out such operations. Users own other instances, for example a user owns experiments or projects that they create. Ownership is both an auditing construct - the testbed can determine which researcher configured which objects - and through policy assigns some rights to carry out operations. The owner of an experiment has rights outside those specified by the circles the owner belongs to.

Each user has a unique identifier, a textual userid. Permissions, resources, objects, etcetera are all bound to that userid. A userid is a text string that uniquely identifies a user. A userid cannot contain the colon character (:) nor can it conflict with other userids or projectids.

A user creates a namespace for circles, experiments, and libraries. Instances of each of those abstractions are names as namespace:name. A user with userid u can create experiments with names of the form u:experiment. This allows each user to manage their own names without introducing ambiguity when another user creates an experiment with the same name. Applications are welcome to hide the prefix on such names, except when needed to resolve an ambiguity. In such a case, the application may display only the parts of experiment names after the colon to user u unless that user also has access to another user's experiment with the same name after the colon. Such an application might display an experiment list to user u that looks like:

<table>
<thead>
<tr>
<th>Experiment Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>myworm</td>
</tr>
<tr>
<td>DDoS</td>
</tr>
<tr>
<td>bob:mytest1</td>
</tr>
<tr>
<td>u:mytest1</td>
</tr>
</tbody>
</table>

Most experiments are unambiguous, but because u can see the experiment that user bob called mytest1 and the experiment that u called mytest1, the application presents the prefixes as well.

## Authentication

Authentication is the problem of binding a request to a userid.

In order for a user to carry out operations on the testbed, the user must log in to the testbed. When requests are made to the SPI they are made through a secure connection where client and server (that is user and testbed) are identified by an X.509 certificate issued by the testbed. The login process
binds an X.509 certificate used to make a secure connection to a userid in the testbed used to determine rights. Users are authenticated through a password system.

The simplest way to log in through the SPI is to call the `requestChallenge` operation on the `Users` service. The caller specifies the sorts of challenges it can carry out and the server sends the input for such a challenge. Challenges can include hashing passwords and other mechanisms. Because the challenge and its response are passed through encrypted channels, we also support a clear challenge. In a clear challenge the caller submits a password in the clear. Each challenge is valid for a limited time - 2 minutes - and has a unique identifier so a response is bound to a specific challenge.

Once the caller gets a challenge, it calculates the response and replies to the challenge by calling `challengeResponse` on the `Users` service. If successful, the user is logged in and can begin carrying out operations.

The login process binds a user to an X.509 certificate, so the certificate and the challenge response need to work together. There are two ways to do this: if the user has no X.509 certificate from the testbed, the `challengeResponse` request can be made without a client certificate. In that case, the result of a successful challenge will include a signed certificate and a private key in PEM format that the user can use for future communications. The key is unencrypted, and it is the responsibility of the client to properly protect it.

If the user has an X.509 certificate - say from a previous successful `challengeResponse` exchange - the user can make the challenge response connection with that certificate, and the certificate/user binding will be complete and no new certificate created.

In either case, connections made with that X.509 client certificate will be treated as from the user until the user logs out using the `logout` operation on the `Users` service, logs in as another user as above, or the login times out in 24 hours.

Some web applications have trouble adding an X.509 certificate on the fly - especially those that run in browsers. To simplify the development of those applications, there is an operation in the `ApiInfo` service called `getClientCertificate` that returns a certificate issued by the testbed and an unencrypted private key, both in PEM format. This interface does not require a login and can be accessed through most web browsers, enabling a user to cut and paste the certificate into a local file for use with applications. The certificate is not logged in.

Additionally, a user can always get the server’s certificate to check or add to a trusted servers file using `getServerCertificate` from the `ApiInfo` service.

Applications will have to carry out these challenges before doing anything else and can choose to log users out as well.

**Profile Management**

The `Users` service supports the operations described in the `profiles` description to allow applications to manipulate user profiles.

**Notifications**

Many operations in the SPI require a user to agree with the operation before the operation becomes final. Users who wish to join a project or a circle request permission from the users who own or administer them. Administrators who wish to add users to their circles request confirmation from the users. Basically any operation that expands the rights of users requires the users in question to concur. In order to let users know when such changes are pending or desired, the SPI supports a system for managing short messages available through the `Users` service.

Notifications are these short messages. Notifications are sent by the system in response to actions requiring confirmation. This is not a general inter-user messaging system.

A notification consists of the text of the message a unique identifier for the message, and a set of flags. The valid flags are:

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Urgent</td>
<td>The message is more important than most messages</td>
</tr>
<tr>
<td>Read</td>
<td>The user has read this message</td>
</tr>
</tbody>
</table>

Each user has a queue of notifications they can access. The `getNotifications` operation on the `Users` service returns an array of notifications for an application to display. Notifications can be filtered by source and by flags. The flags can be set using the `markNotifications` operation that specifies notifications by unique ID.

Applications will use these operations to:

- Present a user with a list of pending notifications when a user logs in, or at regular intervals.
- Display urgent and other notifications differently.
- Allow users to read and respond to notifications by calling various SPI operations to confirm pending actions.
• Search notifications for past actions
• Mark notifications as read once viewed and acted upon

Note that users generally cannot send notifications to other users. Only administrators and testbed actions do this.

Password Management

The Users service provides operations for managing a user’s password. A user who is logged in can change their password directly using the changePassword operation. The operation takes a user identifier and a password. Administrators can also use that interface to change other user’s passwords.

If a user cannot log in - for example their password has expired or they have forgotten it - the application can use the requestPasswordReset interface to issue temporary credentials that can be used to set a user’s password to a known value. When requestPasswordReset is called the credentials are e-mailed to the user’s e-mail address. That address is part of their required profile. The application can then gather a password from the user and call the changePasswordChallenge operation with the new password and the temporary credentials from the e-mail. To make it easier for browser based applications to incorporate this interaction into their operations, requestPasswordReset takes a URL prefix that it can embed in the challenge it sends to the user, allowing the user to click on that URL in the mail to transfer the temporary credentials back to the application.

As an example of using the URL prefix, if a user forgets their password, the application can call requestPasswordChange with a URL prefix of https://myapp.example.com/resetpassword?challenge=. The user will get an e-mail including the link https://myapp.example.com/resetpassword?challenge=temporary_credentials that they can click on. That link will take the user back into the web-based application to a page that collects the new password. The app has the temporary credentials from the URL.

Creating New Users

To create a new account/user instance, an application calls the createUser operation from the Users service. The application must provide a valid profile, can provide a preferred user ID, and an application URL prefix.

The profile must have all non-optional fields present, and any fields having required formats correctly formatted. The application can get the profile requirements from the profile service. Note that all READ_ONLY fields are specified at this time - creation is when they are written.

The application can request a user ID if the user requests one. If that ID is already taken, the testbed will pick a similar one. If no ID is specified, a base user ID is generated from the required e-mail address and if that is taken, a similar one is generated.

The request can also include a URL prefix that works the same way as the URL prefix in the password reset interface. The user is created without a valid password, and the same sort of temporary credentials are issued and mailed to the user. A browser-based application can use the URL prefix to redirect the user back to the application to set their initial password.

Once the user is created through this interface, that user has very few rights because it is not a member of an approved project. It can reset its password and either create a project or attempt to join an existing one.

When a new user is created, that user is temporarily logged in and can issue createProject and joinProject requests. That grace period is currently 30 minutes. After that time elapses, the user must set their password by responding to the e-mail sent by the createUser request.

This grace period is intended to make the user’s experience joining the testbed and creating their first project more seamless. They need not leave the UI to validate their account before making their first createProject or joinProject requests.

Administrative Operations

There are a few more operations that are restricted to administrators.

There are profile operations to add, remove and change attributes to the user profile schema.

There is an operation sendNotification to send a system notification to users.

There is an operation createUserNoConfirm to create a new user instance with a valid password, without going through an e-mail confirmation to set a password.

These are documented below.

Projects

A Project is an administrative grouping of users that helps to scale testbed administration by allowing trusted users to add others to the testbed. Intuitively, a project collects users who are working on a particular research project. A user proposes such a project to the testbed administration. If the
administration approves that project, the proposing user is granted the power to add other users to the project. Until a user is a member of an approved project, they have minimal rights on the testbed.

To make the distinction above explicit, projects can be in 2 states: approved and unapproved. All projects are initially unapproved. Members of unapproved projects gain no additional rights on the testbed. Once an administrator approves the project, the members gain broader rights. Approval is an explicit operation undertaken only in accordance with the testbed's real-world policies.

As users have userids, projects have projectids. A projectid is a text string that uniquely identifies a project. It cannot contain a colon, nor can it conflict with either another projectid or userid.

An approved project creates its own namespace for circles, experiments, and libraries in much the same way creating a user does. Users can be explicitly given the right to create circles, experiments, and libraries in the project's namespace. The SPI places no meaning on an instance created with a user name prefix rather than a project name prefix, but applications may.

A project also is linked to a circle named after the project. A project named myproject will have a linked circle myproject:myproject. The system keeps the membership of that circle and the project synchronized. By assigning rights to that circle, users can assign rights to all project members.

The operations on projects are controlled by the Projects service.

When a researcher joins the testbed by creating a user instance, one of the few operations the testbed will allow them to carry out is to propose a project. The intended workflow is that a researcher comes to the testbed, joins it by creating a user instance and then gains rights by proposing a project that is approved. Then that user (and other users to which it delegates rights) can administer access under that project.

**Project Permissions**

Any member of a project may have one or more of the following permissions that define the operations that user can carry out on the project itself. The permissions are:

<table>
<thead>
<tr>
<th>Permission Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_USER</td>
<td>This user may request that a user be added to the project or confirm a request by a user to join the project</td>
</tr>
<tr>
<td>CREATE_CIRCLE</td>
<td>This user may create a new circle in the project's namespace (a name prepended by projectid)</td>
</tr>
<tr>
<td>CREATE_EXPERIMENT</td>
<td>This user may create a new experiment in the project's namespace (a name prepended by projectid)</td>
</tr>
<tr>
<td>CREATE_LIBRARY</td>
<td>This user may create a new library in the project's namespace (a name prepended by projectid)</td>
</tr>
<tr>
<td>REMOVE_USER</td>
<td>This user may remove a user from the project. Any objects they have created in the project's namespace remain</td>
</tr>
</tbody>
</table>

**Creating A Project**

A user creates a project using the createProject operation on the Projects service. Mechanically, the procedure to create a project is very similar to creating a user. An application will:

- Call the getProfileDescription operation on the Projects service to get an empty project profile
- Gather data from the user to fill in the project description, including non-optional fields and proper formatting
- Call the createProject with a project name from the user and that user as the project's owner. (Administrators can create projects owned by other users).

When createProject completes successfully, the project exists but is unapproved. An administrator can approve it by calling approveProject in the Projects service once the testbed's real-world criteria are met. An administrator may also call removeProject in the Projects service to remove an unapproved project. (removeProject also will remove approved projects, potentially removing many users' rights to operate on the testbed.)

When a new user is created, that user is temporarily logged in and can issue createProject requests. That grace period is currently 30 minutes. After that time elapses, the user must set their password by responding to the e-mail sent by the createUser request.

This grace period is intended to make the user's experience joining the testbed and creating their first project more seamless. They need not leave the UI to validate their account before making their first createProject or joinProject requests.
Adding Users To A Project

Users can become members of a project by requesting to join or by being invited to join. Because joining a project adds a user to the project's circle, that changes the rights of a user, so either of these routes must be endorsed by both the user and someone with relevant permissions in the project. The difference is only the order in which the endorsements are gathered.

When a user finds a project they would like to join, usually through some out of band mechanism, they call the joinProject operation on the Projects service. This registers the user's intention to join and sends a notification to members of the project who have the right to add users to it. This notification includes a challenge that is similar in format to the requestPasswordChange challenge. As with that operation, an application can include a URL prefix that will be prepended to the challenge in the notification.

When a new user is created, that user is temporarily logged in and can issue joinProject requests. That grace period is currently 30 minutes. After that time elapses, the user must set their password by responding to the e-mail sent by the createUser request.

This grace period is intended to make the user's experience joining the testbed and creating their first project more seamless. They need not leave the UI to validate their account before making their first createProject or joinProject requests.

One of the users receiving the notification can then call joinProjectConfirm with the challenge and a list of permissions that the user will be assigned. When that operation completes, the user is a member of the project with the associated rights. The user calling joinProjectConfirm must have the ADD_USER right and similarly cannot confer more rights than they have.

When a user with the ADD_USER permission wants to invite another user to join the project, the inviting user calls the addUser operation with the projectid, a list of users, proposed permissions, and an a URL prefix that serves the same purpose as the prefix on joinProject. When the operation succeeds, each invited user receives a notification containing a challenge (and descriptive text about the project). If the invited user does wish to join, they call the addUserConfirm operation with the challenge. When that operation succeeds, they are a member of the project.

A user with the REMOVE_USER permission can unilaterally remove other users from the project by calling the removeUsers operation. The owner of a project cannot be so removed.

Project Ownership

When a project is created, an owner for that project is designated. It is generally the user who successfully called createProject, though ownership may be designated for projects created by administrators. The owner's identity may be a factor in project approval, depending on how the testbed is administered.

The owner controls the contents of the project profile and is the only project member who can remove the project from the system (though administrators can also do that).

The owner (and administrators) can change the owner of a project by calling the setOwner operation.

Changing Project Permissions

A user that has both ADD_USER and REMOVE_USER permissions to a project can change the permissions granted to a user in the project. The changePermissions operation in the Projects service is used to do this. It takes a projectid, a list of users, and a new set of permissions. It returns an array of results indicating the outcome of each requested change.

Project Profile Management

The Projects service supports the operations described in the profiles description to allow applications to manipulate project profiles. Only the owner of a project can change the contents of its profile.

Removing a Project

The owner of a project (or an administrator) can remove a project using the removeProject operation on the Projects service. This removes the project and profile from the system.

Removing an approved project from the system may revoke access to the testbed from users. A user who was a member of only one approved project will lose their access if that project is removed.

Viewing Projects

A user can always view projects that they are a member of. The viewProjects operation on the Projects service returns summaries of the various projects. Each summary includes the projectid, members, and approval status of the project. Given the projectid, the user can then query the profile.
The `viewProjects` interface takes the user doing the search and an optional regular expression that scopes the search. If the regular expression is given, only projectids matching it are returned. Administrators can use `viewProjects` to get a list of project descriptions visible to a given user.

**Administrative Operations**

There are a few more operations that are restricted to administrators.

There are profile operations to add, remove and change attributes to the project profile schema.

There is an operation, `addUsersNoConfirm` to add users to a project without requiring the users being added to confirm the change.

These are documented below.

**Circles**

Circles are groups of users used to share the contents of experiments and libraries among users. One can think of them as lightweight projects of collaborating researchers who are working together on related experiments.

The key distinction between the two kinds of user groups is that a project confers access to the testbed resources in the large based on delegation of trust. A circle confers rights to specific experiments and libraries. Project membership is a prerequisite to accessing the testbed at all; circle memberships control access to specific abstraction instances inside the testbed.

Once the testbed administration has placed their trust in a user through the project abstraction, they are free to collaborate without any further significant administrative oversight through the circles system.

Projects and circles are related. Each project has a linked circle that contains all the users who are members of the project. The system keeps that circle and project membership synchronized. One can think of assigning the rights to manipulate experiments and libraries to projects as existing Emulab code does. Under the covers it is the linked circle that conveys the rights.

Whenever a user is created, a circle, named `userid:userid`, is also created containing only that user. One can use this circle to assign rights to a user and only that user. This circle behaves like all other circles, but may seem special because of the membership and permissions assigned to it. There is only the one member, and that member does not have the rights to modify the circle membership. The system removes this circle when and if the user is removed.

Finally, each user is a member of the world circle, named ‘system:world’. A user cannot remove themselves from this circle, and it is not returned when `viewCircles` is used to `view circles`. The presence of the circle allows other users to easily make experiments and libraries visible to all other users.

**Circle Names**

Every circle has a unique `circleid` that names it in the testbed. Circle names are scoped by either userids or projectids. A `circleid` has the form of either:

- `userid:local_name`
- `projectid:local_name`

So there may be circles named `some_user:students` and `some_project:students`. They are distinct.

That scoping serves 2 purposes: disambiguating common names and providing a way to distinguish certain names. We expect that many users will set up circles with local names like `test`, `students`, `assistants` and the like. The user interfaces built on the SPI can scope such names by the userid creating them and hide the prefix unless it is necessary to disambiguate.

When a user creates a name in scoped by a project name, it is functionally equivalent to a userid-scoped name, but applications may choose to distinguish them. One can easily imagine that `some_user:students` and `some_project:students` refer to different sets of users and that the project-prefixed circle is administered more carefully and used more widely. The right to create a circle in a project's name space - prefixed by the projectid - is controlled by project permissions.

**Circle Permissions**

Like `projects`, each user in a circle may have one or more of the following permissions that define the operations that user can carry out on the circle itself. The permissions are:

<table>
<thead>
<tr>
<th>Permission Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_USER</td>
<td>This user may request that a user be added to the project or confirm a request by a user to join the project.</td>
</tr>
<tr>
<td>REALIZE_EXPERIMENT</td>
<td>This user may allocate resources to an experiment and carry it out so that it is accessible to the members of this circle.</td>
</tr>
<tr>
<td>REMOVE_USER</td>
<td>This user may remove a user from the project. Any objects they have created in the project's namespace remain.</td>
</tr>
</tbody>
</table>

---

14
The REALIZE_EXPERIMENT permission means that a user can carry out an experiment under this circle. This means members of the circle will:

- have access to the running experiment - e.g., accounts on computers, permissions to connect to orchestration software, etc.
- have access to data collected from the experiment

Creating A Circle

A user creates a circle using the createCircle operation on the Circles service. Mechanically, the procedure to create a project is very similar to creating a user or project. An application will:

- Call the getProfileDescription operation on the Circles service to get an empty circle profile
- Gather data from the user to fill in the circle profile, including non-optional fields and proper formatting
- Call the createCircle with the profile, a circle name from the user, and that user as the project’s owner. (Administrators can create projects owned by other users).

Unlike when a user creates a project, when createCircle completes successfully, the circle is ready for use.

Adding Users To A Circle

As with projects, users can become members of a circle by requesting to join or by being invited to join. Because joining a circle changes the rights of a user, either of these routes must be endorsed by both the user and someone with relevant permissions in the circle. The difference is only the order in which the endorsements are gathered.

When a user finds a circle they would like to join, usually through some out of band mechanism, they call the joinCircle operation on the Circles service. This registers the user's intention to join and sends a notification to members of the circle who have the right to add users to it. This notification includes a challenge that is similar in format to the addUsers challenge for projects. As with that operation, an application can include a URL prefix that will be prepended to the challenge in the notification.

One of the users receiving the notification can then call joinCircleConfirm with the challenge and a list of permissions that the user will be assigned. When that operation completes, the user is a member of the circle with the associated permissions. The user calling joinCircleConfirm must have the ADD_USER right and similarly cannot confer more rights than they have.

When a user with the ADD_USER permission wants to invite another user to join the project, the inviting user calls the addUsers operation with the circleId, a list of users, proposed permissions, and an a URL prefix that serves the same purpose as the prefix on joinCircle. When the operation succeeds, each invited user receives a notification containing a challenge (and descriptive text about the circle). If the invited user does wish to join, they call the addUserConfirm operation with the challenge. When that operation succeeds, they are a member of the circle.

A user with the REMOVE_USER permission can unilaterally remove other users from the circle by calling the removeUsers operation. The owner of a circle cannot be so removed.

Circle Ownership

When a circle is created, an owner for that circle is designated. It is generally the user who successfully called createCircle, though ownership may be designated for circles created by administrators. Ownership primarily means that the owner cannot be removed from the circle and controls the profile contents.

The owner (and administrators) can change the owner of a circle by calling the setOwner operation in the Circles service.

Changing Circle Permissions

A user that has both ADD_USER and REMOVE_USER permissions to a circle can change the permissions granted to a user in the circle. The changePermissions operation in the Circles service is used to do this. It takes a circleId, a list of users, and a new set of permissions. It returns an array of results indicating the outcome of each requested change.

Circle Profile Management

The Circles service supports the operations described in the profiles description to allow applications to manipulate circle profiles. Only the owner of a circle can change the contents of its profile.

Removing a Circle

The owner of a circle (or an administrator) can remove a circle using the removeCircle operation on the Circles service. This removes the circle and profile from the system.
Project circles, user circles, and the world circle cannot be successfully removed. Project circles are removed when the project is, user circles when the user is, and the world circle when the entire testbed is.

When a circle is removed, any access rights granted to users by virtue of their membership in it are also revoked.

**Viewing Circles**

A user can always view circles that they are a member of. The `viewCircles` operation on the Circles service returns summaries of the various circles. Each summary includes the circleid, as well as the members and their permissions. Given the circleid, the user can then query the profile.

The `viewCircles` operation takes the user doing the search and an optional regular expression that scopes the search. If the regular expression is given, only circleids matching it are returned. Administrators can use `viewCircles` to get a list of circle descriptions visible to a given user.

The system:world circle is never returned by `viewCircles` but all users are members of that circle.

**Administrative Operations**

There are a few more operations that are restricted to administrators.

There are profile operations to add, remove and change attributes to the circle profile schema.

There is an operation, `addUsersNoConfirm` to add users to a circle without requiring the users being added to confirm the change.

These are documented below.

**Experiments**

The experiment abstraction encapsulates the layout, procedure, data requirements, and constraints on an activity a researcher using to play with ideas or test a hypothesis. An experiment is realized by DETERLab into an environment that meets those requirements. Realizing an experiment allocates resources in the testbed and grants access to the members of the circle in which the experiment is realized.

Experiments are the unit of testbed activity that are shared between users for understanding ideas and reproducing results.

The layout, procedure, data requirements and constraints on an experiment are sub-abstractions of an experiment called aspects. An experiment does not necessarily contain all aspects. When a researcher first comes to DETERLab, their models and ideas are likely to be loosely defined. We expect them to realize experiments that have only layout aspects. As their ideas become more concrete and their results more stable, the other aspects become more fleshed out.

We present a worked example of the layout aspect that illustrates this.

Experiments also have profiles attached to them so that other users can understand the purpose and goals of an experiment without investigating the aspects in detail.

**Aspects**

The SPI treats aspects as first class objects that can be directly manipulated. Users can add or remove them from experiments, and aspects interact with one another. One can specify a data gathering or procedure aspect that is inconsistent with the layout aspect, and the SPI will find the discrepancy.

Internally, aspects are plug-in modules of code, written to a well-defined interface, that cooperate to

- Generate experiment realizations
  - Build layouts for experimentation and data gathering
  - Control virtualization and simulation implementations
  - Create layouts in the underlying hardware
- Orchestrate experiments
  - Carry out procedures
  - Collect data from experiments
- Present data
  - Collect datasets
  - Visualize ongoing and completed experiments

To support the ongoing research in all these areas, and in areas we do not know about yet, the aspect interfaces are open, and the SPI implementation can easily import code written to it. We expect to add SPI interfaces to dynamically add an aspect implementation to a running testbed.
Because research on individual aspects is ongoing, the SPI presents fairly generic interfaces to them. Each aspect is specified by a type, sub-type and name. The operation of the aspect is controlled by a data block (often a file) attached to the (type, sub-type, name). The meanings of sub-types, names, and the interpretation of data blocks within aspects are controlled by the aspect.

To allow aspects to continue to be developed, the SPI minimally constrains aspect configuration. The data block passed to an aspect is generally a configuration in the aspect's specific configuration language. These are specified individually for each aspect.

When an aspect is added to an experiment, the SPI starts a transaction across the various aspect implementations present in the experiment and being added to the experiment. Each is given a chance to inspect any new instances of the aspect being added to the experiment and they may query the current experiment contents? looking for any inconsistencies with other aspects. When new instances have been inspected, the transaction is closed, giving all aspects (new and old) a chance to veto the addition. This same transaction protocol is followed when aspects are changed or deleted.

**Layout Aspect**

The layout aspect specifies the elements and interconnections of an experimental environment. We present it in some detail, both because researchers will use it practically and as an example of a fleshed out aspect.

**Experiment Names**

Like circles, experiments are named within the scope of either a user (prefixed by userid) or project (prefixed by projectid). The right to create an experiment in a project is controlled by the `CREATE_EXPERIMENT` project permission.

**Experiment Permissions**

As with other abstractions, access to experiments is controlled by permissions. These permissions are assigned to circles rather than individual users. Because each user is the sole member of an immutable circle named `userid` permissions can be assigned per-user by assigning to that circle. Similarly, assigning to the `projectid` circle assigns permissions to all members of a project.

The permissions relevant to project are:

<table>
<thead>
<tr>
<th>Permission Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODIFY_EXPERIMENT</td>
<td>Circle members can change aspects of an experiment</td>
</tr>
<tr>
<td>MODIFY_EXPERIMENT_ACCESS</td>
<td>Circle members can change the permissions of other circles</td>
</tr>
<tr>
<td>READ_EXPERIMENT</td>
<td>Circle members can read experiment aspects and information about access control</td>
</tr>
</tbody>
</table>

Experiments are realized in circles. A user may realize an experiment - allocate resources to it and set it running - if that user holds the `REALIZE_EXPERIMENT` permission to the circle and the `READ_EXPERIMENT` permission to the experiment.

**Creating A New Experiment**

A user creates a circle using the `createExperiment` operation on the Experiments service. Mechanically, the procedure to create a project is very similar to creating a circle or project, though aspect data is also included. An application will:

- Call the `getProfileDescription` operation on the Experiments service to get an empty experiment profile
- Gather data from the user to fill in the experiment profile, including non-optional fields and proper formatting
- Call the `createExperiment` with an experiment name from the user and that user as the experiment's owner. (Administrators can create projects owned by other users). Additionally, initial aspect definitions and initial access control lists may be specified.

When `createExperiment` returns the experiment can be further manipulated. An experiment with no access control lists and no aspects is a valid experiment. Experiment owners always implicitly have all permissions to the experiment as well as the rights to control ownership and profile contents.

Because we expect some researchers to build some experiments incrementally from scratch, experiments are very liberal about their contents. While an experiment with no access control lists and no aspects is valid, it is not very interesting to realize it until it has some aspects.

A user can also specify all the experiment aspects and access control lists when `createExperiment` is called.

**Adding and Changing Experiment Aspects**

A user that is a member of a circle that has the `MODIFY_EXPERIMENT` permission on an experiment can add or remove aspects from that experiment. The operation to add aspects is `addExperimentAspects` in the Experiments service. Changing existing aspects is done through `changeExperimentAspects`.
The parameters to `addExperimentAspect` are arrays of the type, subtype, name and data block. Each of these are defined by the various aspects. The `layout aspect` defines one set of properties. In addition, adding one aspect may result in additional subaspects being added as well. The `layout aspect` does this as well.

There is a default aspect implementation that simply copies the data block of the `addExperimentAspects` call and stores it. When an otherwise unknown aspect type is requested, this implementation is called. The intent is that service developers can copy data in and out of experiment definitions by picking an aspect type name.

The `changeExperimentAspects` takes the same list of type, sub-type, name, and data block tuples, but in this case the data block is an aspect-specific update language. The intention is to allow aspects to define update protocols. To replace an aspect a user can generally call `removeExperimentAspects` and then `addExperimentAspects` to replace the removed aspects. The `layout aspect` does not define a change protocol. The default aspect implementation simply copies the contents of the data block over the existing data (if any).

**Adding and Changing Experiment Access Control Lists**

All experiment permissions are controlled by access control lists (ACL) that map circles to experiment permissions. The only exception to this is the owner of the experiment who always has all permissions. Any user who has `MODIFY_EXPERIMENT_ACCESS` permissions to an experiment can change its ACLs.

A user can set the ACLs on `createExperiment` by providing a list of (circle, permissions) pairs, where permissions is itself a list of permissions. Any errors in specifying the ACLs will cause the creation to fail.

A user can call `changeExperimentACL` with a list in the same format. New entries will be added, existing entries will be overwritten, and entries with an empty permission list will be removed. A partial success is possible and the result of each change is returned.

**Experiment Ownership**

When an experiment is created, an owner for that experiment is designated. It is generally the user who successfully called `createExperiment`, though ownership may be designated for experiments created by administrators. An owner retains all permissions to manipulate the experiment regardless of ACLs and controls the profile contents.

The owner (and administrators) can change the owner of a experiment by calling the `setOwner` operation in the `Experiments` service.

**Experiment Profile Management**

The `Experiments` service supports the operations described in the profiles description to allow applications to manipulate experiment profiles. Only the owner of an experiment can change the contents of its profile.

**Removing an Experiment**

The owner of an Experiment (or an administrator) can remove an experiment using the `removeExperiment` operation on the `Experiments` service. This removes the experiment and profile from the system. Only unrealized experiments may be removed.

**Viewing Experiments**

A user can always view experiments that they can read. The `viewExperiments` operation on the `Experiments` service returns descriptions of the various experiments. Experiment descriptions contain more information than the descriptions of circles. Specifically aspects and ACLs can be selected individually to be shown for each experiment returned.

The `viewExperiments` operation takes the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td>Return experiments this user can read. Admins may successfully view experiments on another user, but most users can only specify the user id here.</td>
</tr>
<tr>
<td>lib</td>
<td>Restrict returned experiments to those in this library.</td>
</tr>
<tr>
<td>regex</td>
<td>Restrict returned experiments to those that have an experimentid that matches the given regular expression.</td>
</tr>
<tr>
<td>queryAspects</td>
<td>Restrict the output aspects of each experiment description to aspects that match one of the list of tuples given. Each tuple is a (type, sub-type, name) tuple. If the name is not null, it must match. If a type is not null it must also match. If a type is given the sub-type field further narrows the search. A null type field selects aspects without any sub-type. A specific sub-type selects only aspects that match both type and sub-type, and the distinguished value &quot;*&quot; matches any sub-type. A sub-type with a null type is invalid.</td>
</tr>
<tr>
<td>listOnly</td>
<td>If true, do not return aspect data in the response.</td>
</tr>
<tr>
<td>offset</td>
<td>Begin returning experiment descriptions at the position of a global list of matching experiments, ordered by creation time. This allows incremental retrieval of experiment lists.</td>
</tr>
<tr>
<td>count</td>
<td>Return only this many experiment descriptions.</td>
</tr>
</tbody>
</table>
Each description contains the following information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>experiment ID</td>
<td>The unique experiment identifier</td>
</tr>
<tr>
<td>owner</td>
<td>The userid of the owner</td>
</tr>
<tr>
<td>perms</td>
<td>The permissions that the uid can exert on the experiment. This allows an application to report the permissions without calculating them from the circles in the ACL</td>
</tr>
<tr>
<td>ACL</td>
<td>The access control list as a list of (circle, (permissions,... ) ) pairs. Permissions is a list of permissions.</td>
</tr>
<tr>
<td>Aspects</td>
<td>The aspects that match the queryAspects parameter as a list of (type, sub-type, name, data block) tuples. If listOnly was given, the data blocks will be empty.</td>
</tr>
</tbody>
</table>

**Administrative Operations**

There are a few more operations that are restricted to administrators.

There are profile operations to add, remove and change attributes to the experiment profile schema.

These are documented below.

**Libraries**

Libraries are groups of experiments used to collect related experiments. For example, a testbed may have a library of experiments for new users to realize and study in learning to use the testbed. A research group might collect a set of regression tests into a library.

As with experiments, access to the library is controlled by circles. A library if beginner experiments might be readable by the world circle while a specific research group would protect their libraries using project circles or smaller.

Libraries also have profiles attached to them so that other users can understand the purpose and goals of the library.

**Library Names**

Every circle has a unique libraryid that names it in the testbed. Library names are scoped by either userids or projectids. A libraryid has the form of either:

- userid:local_name
- projectid:local_name

So there may be libraries named some_user:experiments and some_project:experiments. They are distinct.

That scoping serves 2 purposes: disambiguating common names and providing a way to distinguish certain names. We expect that many users will set up libraries with local names like test, development, paper_results and the like. The user interfaces built on the SPI can scope such names by the userid creating them and hide the prefix unless it is necessary to disambiguate.

When a user creates a name in scoped by a project name, it is functionally equivalent to a userid-scoped name, but applications may choose to distinguish them. One can easily imagine that some_user:development and some_project:development refer to different sets of experiments and that the project-prefixed library is administered more carefully and used more widely. The right to create a library in a project’s name space - prefixed by the projectid - is controlled by project permissions.

**Library Permissions**

As with other abstractions, access to libraries is controlled by permissions. These permissions are assigned to circles rather than individual users. Because each user is the sole member of an immutable circle named userid:userid permissions can be assigned per-user by assigning to that circle. Similarly, assigning to the projectid:projectid circle assigns permissions to all members of a project.

The permissions relevant to project are:

<table>
<thead>
<tr>
<th>Permissions Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_EXPERIMENT</td>
<td>Circle members can add experiments to a library</td>
</tr>
<tr>
<td>MODIFY_LIBRARY_ACCESS</td>
<td>Circle members can change the permissions of other circles</td>
</tr>
<tr>
<td>READ_LIBRARY</td>
<td>Circle members can read experiment names and information bout access control from the library</td>
</tr>
<tr>
<td>REMOVE_EXPERIMENT</td>
<td>Circle members can remove experiments from the library</td>
</tr>
</tbody>
</table>
A library is little more than a list of experiments. The rights to inspect, modify, or realize experiments in a library are completely controlled by the permissions that a user has to the experiment. Libraries provide a way to let users know that experiments exist and are related, not to control the experiments.

**Creating A Library**

A user creates a library using the `createLibrary` operation on the Libraries service. Mechanically, the procedure to create a library is very similar to creating a user or project. An application will:

- Call the `getProfileDescription` operation on the Libraries service to get an empty library profile
- Gather data from the user to fill in the library profile, including non-optional fields and proper formatting
- Call the `createLibrary` with the profile, a libraryid name from the user, and that user as the library's owner. (Administrators can create projects owned by other users). Other parameters include an initial list of experimentids in the library and initial access control lists.

When `createLibrary` completes successfully, the library is ready for use.

**Adding Experiments To A Library and Removing Them**

The operations to add and remove experimentids from a library are pretty straightforward: `addLibraryExperiments` adds new ones and `removeLibraryExperiments` (both in the Libraries service) do the obvious things. Both take a list of experimentids and return a result for each addition or removal. `addLibraryExperiments` requires the user to have the `ADD_EXPERIMENT` permission; `removeLibraryExperiments` requires `REMOVE_EXPERIMENT`.

**Adding and Changing Library Access Control Lists**

All library permissions are controlled by access control lists (ACL) that map circles to library permissions. The only exception to this is the owner of the library who always has all permissions. Any user who has `MODIFY_LIBRARY_ACCESS` permissions to an library can change its ACLs.

A user can set the ACLs on `createLibrary` by providing a list of (circle, permissions) pairs, where permissions is itself a list of permissions. Any errors in specifying the ACLs will cause the creation to fail.

A user can call `changeLibraryACL` with a list in the same format. New entries will be added, existing entries will be overwritten, and entries with an empty permission list will be removed. A partial success is possible and the result of each change is returned.

**Library Ownership**

When an library is created, an owner for that library is designated. It is generally the user who successfully called `createLibrary`, though ownership may be designated for libraries created by administrators. An owner retains all permissions to manipulate the library regardless of ACLs and controls the profile contents.

The owner (and administrators) can change the owner of a experiment by calling the `setOwner` operation in the Libraries service.

**Library Profile Management**

The Libraries service supports the operations described in the profiles description to allow applications to manipulate library profiles. Only the owner of an library can change the contents of its profile.

**Removing a Library**

The owner of a library (or an administrator) can remove a library using the `removeLibrary` operation on the Libraries service. This removes the library and profile from the system.

Removing a library has no effect on the experiments in it.

**Viewing Libraries**

A user can always view libraries that they can read. The `viewLibraries` operation on the Libraries service returns descriptions of the various libraries. Library descriptions contain experimentid lists and ACLs.

The `viewLibraries` operation takes the following parameters:
Each description contains the following information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>library ID</td>
<td>The unique library identifier</td>
</tr>
<tr>
<td>owner</td>
<td>The userid of the owner</td>
</tr>
<tr>
<td>perms</td>
<td>The permissions that the uid can exert on the library. This allows an application to report the permissions without calculating them from the circles in the ACL</td>
</tr>
<tr>
<td>ACL</td>
<td>The access control list as a list of (circle, (permissions,...) ) pairs. Permissions is a list of permissions.</td>
</tr>
<tr>
<td>Experiments</td>
<td>The experimentids in the library</td>
</tr>
</tbody>
</table>

### Administrative Operations

There are a few more operations that are restricted to administrators.

There are profile operations to add, remove and change attributes to the circle profile schema.

These are documented below.

### Other Services

The ApiInfo and Admin services do not directly manipulate the major abstractions. ApiInfo is useful to many applications while Admin is mostly useful for testbed administrators.

#### ApiInfo

The ApiInfo service provides versioning information and some basic application connectivity tests. It also provides a simple mechanism to get server and client certificates from the system.

The getVersion operation does not require any login and returns the version and patch level of the implementation running on this server. If the caller provided an X.509 client certificate, the RFC5280-defined identifier of the certificate is also returned.

The echo operation takes and returns a single string argument. It can be used to confirm that simple SOAP-based parameter passing is working. The echo service does not require a login.

The getServerCertificate operation returns the X.509 certificate that the server is using to authenticate itself. An application may need this to make it part of a trust chain. The certificate is returned as a PEM-encoded string.

The getClientCertificate operation returns a certificate and an unencrypted private key that an application can use as an identity. This is essentially the same kind of certificate returned when a user logs in without a client certificate. The operation takes a parameter that is recorded as the common name in the X.509 certificate returned. Both the key and the certificate are returned in PEM format.

The certificate retrieval operations are simple enough that they can often be carried out through a web browser without using SOAP at all.

#### Admin

The Admin service provides a few operations useful to administrators.

The addPermission call installs a new permission into the system. This is only done when adding new functionality to the system or initially creating a server. It takes 2 parameters, the permission name and the kind of object it is valid for.

The bootstrap call creates a basic system from scratch. If there is no admin user nor a project called admin, such a user, deterboss is created in the admin project. deterboss owns admin and admin is approved. The world circle and a few projects needed for regression tests are also created. The password for the deterboss user is returned. From here an application can build a server using the operations above.

The clearCredentialCache and resetAccessControl calls change the system’s policy. In the first case, any cached policies are removed from memory. In the second case all the cached policies are removed and all policies system-wide are recreated. These are rarely used operations.

#### Detailed Specifications
The current interface specifications are available for both the [Stable version](#) and [Development version](#).

**Stable Version**

- Javadocs of services
- WSDL for service invocation
- XSD for topdl
- SOAP traces of regression tests (most will not need these)

**Development Version**

- Javadocs of services
- WSDL for service invocation
- XSD for topdl
- SOAP traces of regression tests (most will not need these)