Author:
Last Modified: 2004-10-02T20:12:00Z
By:

Table of Contents

1 Overview..... 4

1.1 API Requirements..... 4

1.2 Other Service Broker Requirements..... 5

2 Design Issues..... 5

2.1 Locating the Abstraction Boundary..... 5

2.2 Authorization and the Effective User Group..... 5

2.3 Lab Clients, Servers, and "Labs"..... 6

2.4 Choosing Among Multiple Lab Servers..... 6

2.5 Data Storage..... 6

2.6 Caching on the Service Broker..... 7

2.7 Location and Maintenance of Client Code..... 7

2.8 Result Notification..... 7

2.9 Experiment Receipts..... 7

2.10 Downloading Data..... 8

2.11 Accessing the Experiment Log..... 10

3 Usage Scenarios..... 11

4 Methods..... 13

4.1 Pass-through Methods..... 13

4.1.1 Cancel..... 13

4.1.2 GetEffectiveQueueLength 13

4.1.2.1 public struct waitEstimate..... 14

4.1.3 GetExperimentStatus..... 14

4.1.3.1 public struct LabExperimentStatus..... 14

4.1.3.2 public struct ExperimentStatus..... 15

4.1.4 GetLabConfiguration 15

4.1.5 GetLabInfo..... 16

4.1.6 GetLabStatus..... 16

4.1.6.1 public struct LabStatus..... 16

4.1.7 RetrieveResult..... 16
4.1.7.1 public struct ResultReport..... 17
4.1.8 Submit..... 18
4.1.8.1 public struct ClientSubmissionReport..... 18
4.1.9 Validate..... 19
4.1.9.1 public struct ValidationReport..... 19
4.2 Client Methods..... 20
4.2.1 DeleteClientItem..... 20
4.2.2 ListAllClientItems..... 20
4.2.3 LoadClientItem..... 20
4.2.4 SaveClientItem..... 21
4.2.5 RetrieveSpecification..... 21
4.2.6 RetrieveLabConfiguration..... 22
4.2.7 SaveAnnotation..... 22
4.2.8 RetrieveAnnotation..... 22
4.2.9 GetExperimentInformation..... 23
5 .....
6 .....

Overview

This document describes the interface between the Lab Client and the Service Broker, and touches on the more general topic of communication between the *user* and the Service Broker. The distinction is made clear by the following definition:

A *Lab Client* (sometimes just "client") is the specialized, domain-dependent piece of software that the user directly interacts with in order to create experiment specifications, submit them for execution (via the service broker), and analyze the results. Not everything the user does is done through the Lab Client; some of it, such as the initial login process, is done through a standard web browser (before the Lab Client is even loaded).

API Requirements

The bulk of this API consists of pass-through methods, whose function is simply to call a corresponding method from the Lab Server API (the Lab Server API is described in a separate document). The pass-through methods are:

- **`GetStatus`**
- **`GetEffectiveQueueLength`**
- **`GetLabInfo`**
- **`GetLabConfiguration`**
- **`Validate`**
- **`Submit`**
- **`Cancel`**
- **`GetExperimentStatus`**
- **`RetrieveResult`**

In addition, the Client to Service Broker API should provide the following functionality:

- Arbitrary opaque data storage (for UI preferences, unfinished experiment setups, etc).
- Access to the user's experiment log (for result retrieval, loading old experiment setups, etc.). Optionally, we may also allow the user to annotate each entry of the experiment log with a comment string (e.g. "6.012 pset 1 problem 5") for ease of later retrieval.

Other Service Broker Requirements

In addition to the core API that is the focus of this document, the Service Broker must provide the following functionality to the user:

- User login
- Loading a Lab Client
- Mechanism for downloading data to be saved as a file on the client machine; discussed in detail on page 8 ()

Design Issues

Locating the Abstraction Boundary

The first challenge in specifying a Client to Service Broker API is that there are potentially many different technologies upon which clients can be built. To ensure that the iLab software infrastructure remains vendor and language neutral, we are specifying the Client/Service Broker API in terms of web services.

Previous versions of the MIT Microelectronics Weblab have used a Java applet as a client. This applet has communicated with an older analogue of the Service Broker using a protocol based on HTTP and ASP web pages. This approach will be our starting point for the first WebLab client built on top of the new Service Broker infrastructure. In this context, the client requires the API specification to be implemented as a Java class that serves as a remote proxy for the ServiceBroker (**`ServiceBroker.java`**). The rest of the Lab Client code is written and compiled against this proxy class. In this case, once the client side proxy class is written in Java, the client can ignore all details of the communication with the Service Broker.

However, this is but one approach. We expect many future Lab Clients to be written as .NET applications in C#. The iLab Project may eventually supply multiple proxies that tailor this API for particular client platforms, as **`ServiceBroker.java`** would for a Java client.

Authorization and the Effective User Group

It is occasionally desirable for users to temporarily limit their own privileges so that, for example, a professor can log in as if he were a student to make sure that everything is in order. Because privileges are assigned based on user groups, most of the ways to do this involve temporarily designating one of the groups that a user belongs to as

his effective user group. A user's privileges are then determined based on his effective user group (rather than based on all of his user groups).

The effective user group is fixed at login time and placed in the session state. It cannot be changed during the current session.

Lab Clients, Servers, and "Labs"

This section attempts to answer the following questions:

- What is the nature and cardinality of the mapping between Lab Clients and Lab Servers?
- What do we mean when we refer to a "lab"? More specifically, when a user logs into the service broker and selects a "lab" to work on, what precisely is being selected?

We have chosen a "Many to several" (static) model for mapping Lab Clients to Lab Servers.

Each Lab Client is designed to use a particular set of Lab Servers (e.g. the MIT Microelectronics Lab Client uses both the MIT Microelectronics Experiment Lab Server and the MIT Microelectronics Device Simulation Lab Server – note that these need not have the same type).

The user chooses a lab by choosing a client. Hence, for our purposes, a lab is defined as a Lab Client.

Choosing Among Multiple Lab Servers

Since a single service broker may provide its users with access to many lab servers, a mechanism is needed for determining which lab server a pass-through method call (such as Submit) should be "forwarded" to.

This will be accomplished by invoking each pass-through method with an extra parameter that identifies the lab server the method call should go to.

Data Storage

The client API provides an opaque data store that functions like a hash table of "attributes" with string names and string values. To avoid conflicts, each (user, client) pair has its own namespace for attributes.

Caching on the Service Broker

In general, information about the current status and configuration of a lab server will not be cached on the Service Broker, at least as far as the Client to Service Broker API calls are concerned. It is essential that when a client calls any of the pass-through methods that retrieve such information, the most up-to-date information is retrieved. The reason for this is that if the API ever returned stale data, the client would have no way to force a refresh (unless we added more methods to the API specifically for this purpose, which would be cumbersome) and the user might be left unable to perform experiments.

There is one possible exception to this: the Service Broker may choose to cache status information for uses other than implementing the API call (e.g. displaying lab status on a web page when the user is choosing a lab), as long as the actual API call will always result in a refresh.

Location and Maintenance of Client Code

Since there may be many clients that use the same lab server, and since one client may use several lab servers, it has been determined that the client software should live on the Service Broker. This necessitates some sort of arrangement between the Service Broker administrators and the client developers to allow for propagation of updated versions of client software (note that this applies equally to clients developed by the same lab staff that runs a lab server and to third-party clients), but this arrangement can easily be made outside the scope of the software infrastructure.

Result Notification

Asynchronous result notification via web service calls from the service broker to the client is not generally an option due to the possibility that the client may be behind a firewall or otherwise not publicly visible, the client may go offline entirely, etc. As a cheap but effective substitute, our API will allow users the option to receive an email notification when an experiment completes. We anticipate that this will be useful primarily for lab domains with experiment runtimes that are long compared to the attention span of a user (so that the user will not want to simply sit and wait for the results to come back).

Experiment Receipts

The client has a contract with the service broker to make sure that an experiment gets executed. The service broker is responsible for subcontracting the job to the lab server, checking on the status of the job as necessary, retrieving the results from the lab server before they are purged, and storing the results for the user to analyze later.

This functionality is precisely the sort of thing that makes our multi-tiered architecture useful. The user can submit an experiment, log off, and come back much later, trusting that his experiment will not have already been purged due to limited lab server resources. At the same time, the lab server is free to maintain a policy which keeps data around just long enough for the service broker (an automated system) to reasonably be able to retrieve it, without having to worry about individual human users who may decide to wait a week before logging back in to retrieve their results.

Note that once an experiment has been submitted, the Service Broker returns a receipt for the experiment in the **`experimentID`** field of the **`SubmissionReport`** object returned by the **`Submit()`** method. This ID is generated by the Service Broker and is guaranteed to be unique. The Service Broker presents the ID to the Lab Server along with the experiment specification when the experiment is originally submitted. The **`experimentID`** identifies a particular experiment throughout the distributed iLab system.

If a client is performing an experiment against more than one Lab Server, the experiment will conceptually be decomposed into separate sub experiments for each Lab Server. Each sub experiment will be submitted separately and identified by its own **`experimentID`**. In effect, as far as the Service Broker and the respective Lab Servers are concerned, these sub experiments will be treated and recorded as full experiments that are only related by the client. At least initially, the client must note the persistent record of this relationship in the annotation portion of the experiment log.

Downloading Data

One desirable feature in many lab clients is the ability to save data to a local file on the user's machine for later external processing by some other application (e.g. a spreadsheet program). However, certain types of clients (such as Java applets) may not be permitted to write files to the user's disk, on the very reasonable grounds that giving file system access to a piece of software downloaded from the

web is a security risk. Essentially, we have two conflicting requirements here: the need to download client code at runtime from a server while preserving client-side security and the need to allow students to save data on the client system.

This is largely (but not entirely) a client-side implementation issue. If the client is a self-standing application, the problem does not exist. The traditional workaround for this problem when the client is an applet involves directing the user's web browser (a piece of application software which is allowed to save files locally) to a file on the server which has been dynamically created by the client. The scenario is as follows:

1. Client sends desired file contents to the server via HTTP POST.
2. Server returns a "magic key" to client.
3. Client directs user's browser to open a URL (via HTTP GET) that includes the "magic key" as a parameter.
4. Server recognizes the "magic key" and returns the desired file contents.

Result: user's browser downloads a file created by the client (and subsequently prompts the user to save that file locally). At first glance, it may seem that there should be a way to do this in a single request, without the intermediate steps, but:

- While a Java applet itself can perform both POSTs and GETs, it can only direct the user's browser to perform a GET (not a POST).
- It is risky to encode the actual file contents in the parameters of a GET, because some web browsers and web servers place arbitrary limits on the maximum URL length they will handle (2,083 characters for Internet Explorer) and the data file might be too big to fit within these limits.

A possible reaction to this procedure might be to observe that since the experiment results are already on the service broker, why do they need to be sent to the client and then sent back to the service broker before they can be downloaded? Why not just let the user's browser download the data that's already there? The reason is that the data on the service broker is an opaque, domain-dependent blob which neither the service broker nor the local spreadsheet program

understands. Only the client knows how to process the data and rewrite it in some other format (e.g. comma-separated value) that will be useful to the spreadsheet program. More generally, the client might also want to save files that contain something else besides experiment results.

Accessing the Experiment Log

The experiment log on the service broker contains a variety of information for each experiment. In addition to the experiment specification, results, and a copy of the lab configuration as of the submission time of the experiment (all of which are domain-dependent and hence opaque to the Service Broker), the experiment log entry may include the following transparent items:

- **`experimentID`** (uniquely identifies this experiment to the service broker and lab server)
- **`userID`**
- Effective group
- **`labServerID`**
- **`priorityHint`**
- Various timestamps (submission, execution beginning and end, result retrieval from lab server)
- Experiment status/completion code
- Warning messages
- Error messages
- Experiment metadata (transparent attribute-value pairs defined by the lab server to help a user identify this experiment)
- Annotation (a comment written by the user to identify this experiment)

Ideally, we would like to have an API that supports sophisticated queries (probably in an XML form) over any and all of this information. However, this is a hard problem which is not immediately crucial to the success of the project, and so we have chosen to simplify the requirements for the first version of the service broker.

We expect that most of the information associated with an experiment will be of an easily manageable size. The notable exception to this is

the experiment results, which might be extremely large (especially if some or all of the data consists of images). The experiment specification and lab configuration are also open to question; probably a single one of these is small, but retrieving a lot of them all at once (in response to a query) may be undesirable. Therefore, our first-version API for accessing the experiment log will present the following functionality:

- Retrieve all data (except for the experiment specification, results, and lab configuration) for entries with a submission time that falls within a specified interval.
- Optionally restrict the results of above to entries with a specified username and/or a specified effective group (by default, you get back all the entries that you have permission to look at, regardless of username or effective group).
- Optionally restrict the results of above to entries with a user annotation that contains a specified string.
- Optionally restrict the results of above by specifying a maximum number of entries to return. If there are more, you get back the ones with the earliest submission times.
- Retrieve the experiment specification, results, and/or lab configuration for a single entry with a specified **`experimentID`**.

All methods that access the experiment log are subject to the access control policy defined by the service broker; if you don't have permission to look at a particular entry, it will not be returned to you. Most users will only have permission to look at their own experiments, but some users (e.g. professors, teaching assistants, and administrators) may have additional permissions.

The client methods to perform these searches will be specified in detail after the *Service Broker Experiment Storage API* is finalized.

Usage Scenarios

The following is a typical usage scenario.

1. User logs into Service Broker website with username and password.
2. Service Broker enumerates user's authorized groups.
3. User selects an effective group (one of the authorized groups).

4. Service Broker enumerates Lab Clients available to the user.
5. User loads a lab client.
6. Client reads user's preferences data from the opaque data store.
7. Client calls **`GetLabConfiguration`**.
8. User creates an experiment specification.
9. Client calls **`Verify`**.
10. Client calls **`Submit`**.
11. Client polls using **`GetExperimentStatus`**.
12. When the experiment is done, client calls **`RetrieveResult`**.
13. Client displays results to user.
14. User closes client and logs off from Service Broker.

Methods

Pass-through Methods

These methods may also be found in the *Service Broker to Lab Server API.*

Cancel

Purpose:

/* Cancels a previously submitted experiment. If the experiment is already running, makes best efforts to abort execution, but there is no guarantee that the experiment will not run to completion. */

Arguments:

`int experimentID`
/* A token that identifies the experiment. */

Returns:

`bool cancelled`
/* **`true`** if experiment was successfully removed from the queue (before execution had begun). If **`false`**, user may want to call **`GetExperimentStatus()`** for more detailed information. */

GetEffectiveQueueLength

Purpose:

/* Checks on the effective queue length of the lab server.

Answers the following question: how many of the experiments currently in the execution queue would run before the new experiment? */

Arguments:

`string labServerID`
/* A token that identifies the lab server. */

`int priorityHint`
/* Indicates a requested priority for the hypothetical new experiment. Possible values range from 20 (highest priority) to -20 (lowest priority); 0 is normal. Priority hints may or may not be considered by the lab server. */

Returns:

waitEstimate WaitEstimate

public struct waitEstimate

{

public int effectiveQueueLength

/* Number of experiments currently in the execution queue that would run before the hypothetical new experiment. */

public double estWait

/* [OPTIONAL, < 0 if not supported] estimated wait (in seconds) until the hypothetical new experiment would begin, based on the other experiments currently in the execution queue. */

}

GetExperimentStatus

Purpose:

/* Checks on the status of a previously submitted experiment. */

Arguments:

int experimentID

/* A token that identifies the experiment. */

Returns:

LabExperimentStatus experimentStatus

/* See description below. */

public struct LabExperimentStatus

{

public ExperimentStatus statusReport

/* See description below. */

public double minTimeToLive

/* Guaranteed minimum remaining time (in seconds) before this **experimentID** and associated data will be purged from the lab server. */

}

public struct ExperimentStatus

/* indicates the status of this experiment. */

{

public int statusCode

/*
1 if waiting in the execution queue,
2 if currently running,
3 if terminated normally,
4 if terminated with errors (this includes cancellation by user in mid-execution),
5 if cancelled by user before execution had begun,
6 if unknown **experimentID**.
*/

public WaitEstimate wait

/* Described on P. 14. */

public double estRuntime

/* [OPTIONAL <0 if not used] estimated runtime (in seconds) of this experiment. */

public double estRemainingRuntime

/* [OPTIONAL < 0 if not used] estimated remaining runtime (in seconds) of this experiment, if the experiment is currently running. */

}

GetLabConfiguration

Purpose:

/* Gets the configuration of a lab server. */

Arguments:

string labServerID

/* A token identifying the lab server. */

Returns:

string labConfiguration

/* An opaque, domain-dependent lab configuration. */

GetLabInfo

Purpose:

/* Gets general information about a lab server. */

Arguments:

`string labServerID`
/* A token identifying the lab server. */

Returns:

`string URL`
/* A URL to a lab-specific information resource, e.g. a lab information page. */

GetLabStatus

Purpose:

/* Checks on the status of the lab server. */

Arguments:

`string labServerID`
/* A token that identifies the lab server. */

Returns:

`LabStatus labStatus`
/* See description below. */

`public struct LabStatus`
{

`public bool online`
/* **`true`** if lab is accepting experiments. */

`public string labStatusMessage`
/* Domain-dependent human-readable text describing status of lab server. */

}

RetrieveResult

Purpose:

/* Retrieves the results from (or errors generated by) a previously submitted experiment. */

Arguments:

int experimentID
/* A token identifying the experiment. */

Returns:

ResultReport resultReport
/* See description below. */

public struct ResultReport

{

public int statusCode
/* Indicates the status of this experiment.
1 if waiting in the execution queue,
2 if currently running,
3 if terminated normally,
4 if terminated with errors (this includes cancellation by user in mid-execution),
5 if cancelled by user before execution had begun,
6 if unknown **experimentID**.
*/

public string experimentResults
/*
[REQUIRED if **experimentStatus** == 3,
OPTIONAL if **experimentStatus** == 4]
an opaque, domain-dependent set of experiment results. */

public string xmlResultExtension
/* [OPTIONAL, null if unused] a transparent XML string that helps to identify this experiment. Used for indexing and querying in generic components which can't understand the opaque **experimentSpecification** and **experimentResults.**/

public string xmlBlobExtension
/* [OPTIONAL, null if unused] a transparent XML string that helps to identify any blobs saved as part of this experiment's results. */

public string[] warningMessages

/* Domain-dependent human-readable text containing non-fatal warnings about the experiment including runtime warnings. */

public string errorMessage
/* [REQUIRED if **experimentStatus** == 4] domain-dependent human-readable text describing why the experiment terminated abnormally including runtime errors. */

}

Submit

Purpose:

/* Submits an experiment specification to the lab server for execution. */

Arguments:

string labServerID
/* A token identifying the Lab Server. */

string experimentSpecification
/* An opaque, domain-dependent experiment specification. */

int priorityHint
/* Indicates a requested priority for this experiment. Possible values range from 20 (highest priority) to -20 (lowest priority); 0 is normal. Priority hints may or may not be considered by the lab server. */

bool emailNotification
/* If **true**, the service broker will make one attempt to notify the user (by email to the address with which the user's account on the service broker is registered) when this experiment terminates. */

Returns:

ClientSubmissionReport clientSubmissionReport
/* See Description Below. */

public struct ClientSubmissionReport
{

public ValidationReport vReport
/* See description below, under . */

public int experimentID
/* The experiment ID. */

public double minTimeToLive
/* Guaranteed minimum time (in seconds, starting now) before this **experimentID** and associated data will be purged from the lab server. */

public WaitEstimate wait
/* See description on P. 14 */

}

Validate

Purpose:

/* Submits an experiment specification to the lab server for execution. */

Arguments:

string labServerID
/* A token identifying the lab server. */

string experimentSpecification
/* An opaque, domain-dependent experiment specification. */

Returns:

ValidationReport validationReport
/* See description below. */

public struct ValidationReport

{

public bool accepted
/* **true** if the experiment specification would be (is) accepted for execution. */

public string[] warningMessages
/* Domain-dependent human-readable text containing non-fatal warnings about the experiment. */

public string errorMessage

/* [If accepted == **false**] domain-depend human-readable text describing why the experiment specification would not be accepted. */

public double estRuntime
/* [OPTIONAL, < 0 if not supported] estimated runtime (in seconds) of this experiment. */

}

Client Methods

DeleteClientItem

Purpose:

/* Removes a client item from the user's opaque data store. */

Arguments:

string name
/* The name of the client item to be removed. */

Returns:

void none

ListAllClientItems

Purpose:

/* Enumerates the names of all client items in the user's opaque data store. */

Arguments:

none

Returns:

string[] clientItems
/* An array of client items. */

LoadClientItem

Purpose:

/* Returns the value of a client item in the user's opaque data store. */

Arguments:

string name
/* The name of the client item whose value is to be returned. */

Returns:

string clientItemValue
/* The value of a client item in the user's opaque data store. */

SaveClientItem

Purpose:

/* Sets a client item value in the user's opaque data store. */

Arguments:

string name
/* The name of the client item whose value is to be saved. */

string itemValue
/* The value that is to be saved with **name**. */

Returns:

void none

RetrieveSpecification

Purpose:

/* Retrieves a previously saved experiment specification.*/

Arguments:

int experimentID
/* A token which identifies the experiment. */

Returns:

string experimentSpecification
/* The experimentSpecification, a domain-dependent experiment specification originally created by the Lab Client. */

RetrieveLabConfiguration

Purpose:

/* Retrieves a previously saved lab configuration for a particular experiment.*/

Arguments:

`int experimentID`
/* A token which identifies the experiment. */

Returns:

`string labConfiguration`
/* The labConfiguration, an opaque string included in the ResultReport by the Lab Server to specify the configuration in which an experiment is executed. */

SaveAnnotation

Purpose:

/* Saves or modifies an optional user defined annotation to the experiment record.*/

Arguments:

`int experimentID`
/* A token which identifies the experiment. */

`string annotation`
/* The annotation to be saved with the experiment. */

Returns:

`string previousAnnotation`
/* The previous annotation or null if there wasn't one. */

RetrieveAnnotation

Purpose:

/* Retrieves a previously saved experiment annotation.*/

Arguments:

`int experimentID`
/* A token which identifies the experiment. */

Returns:

`string annotation`
/* The annotation, a string originally created by the user via the Lab Client. */

GetExperimentInformation

Purpose:

/* Retrieves experiment metadata for experiments specified by an array of experiment IDs.*/

Arguments:

`int experimentID[]`
/* An array of tokens which identify experiments. */

Returns:

`ExperimentInformation[] experimentInformations`
/* An array of instances of the specified ExperimentInformation objects. */