Author:	
Last Modified:	2004-10-02T20:58:00Z
By:	

Table of Contents

Introduction

This design builds upon the basic design concept suggested by the WebLab team that an Internet accessible laboratory be implemented as two independent server processes:

1) the **Lab Server**

   1).)a interfaces to the lab equipment;

   1).)b implements domain and lab specific protocols, e.g. to validate an experiment specification;

   1).)c monitors the performance of the lab equipment;

   1).)d provides a sufficient administrative interface for lab personnel so that they can manage the laboratory equipment;

2) the **Service Broker**

   2).)a authenticates a student using a standard browser interface, and allows the student to perform limited administrative functions such as searching for previous executed experiment results;

   2).)b uploads appropriate experiment-dependent client software to a student's workstation to allow the student to edit, submit, and examine results from experiments; the Service Broker acts as an intermediary between the student's client application and the Lab Server; communication between the student's client and the Service Broker will use web services;

   2).)c authenticates administrators and allows them to register lab servers, manage student accounts, and purge/archive experimental data using web services and a standard browser interface;

   2).)d communicates with one or more lab servers using web services (WS) to forward student experiment specifications and to retrieve results;

This document addresses the category of services 2d above for the category of *batched experiments*. In our emerging typology, a *batched experiment* is one that can be submitted as a single specification and then run without further interaction with the user. Results are returned to the user after the experiment has terminated. WebLab (http://weblab.mit.edu) is a classic example of the batched experiment. In this draft specification, an experiment designates a specific batched use of the laboratory apparatus. An experiment commences when an experiment specification is accepted by the Lab Server and terminates either by normal completion or runtime failure at a point determined by the Lab Server.

This document only specifies the application programming interface (API) by which the Service Broker and Lab Server communicate. The internal APIs by which the browser administrative interfaces are implemented will be described in four further specifications (*Service Broker Experiment Storage API, Service Broker Administrative API, Service Broker Authorization API, Service Broker Authentication API*).

Notes on Return Values

The web service methods described in the next section frequently must return multiple values. We have created a number of fairly arbitrary types to implement these sets of multiple return values as single objects. The types are described under the first method that returns a value of that type. Cross references are given under later methods that return the same type.

These return types frequently contain immutable copies of server data or tokens, e.g., **`experimentID`**. When these return values occur as part of a larger type we have used the access modifiers **`public readonly`** to stress that these fields and the return types as a whole are immutable. Any objects returned are serializable, and they do not contain any useful methods. In fact, we have chosen to specify them as C# **`structs`** to emphasize that they are conceptually value types, not reference types.

Finally, a note on the **`experimentID`** argument to the **`Submit()`** method. This ID, currently an **`int`**, is intended to identify an experiment uniquely within the context of a cooperating Service Broker and Lab Server. The Service Broker guarantees the uniqueness of **`experimentID`** within its own context. A Lab Server may handle experiment submissions from multiple Service Brokers, however, and the **`experimentID`** alone cannot be guaranteed to be unique on such a Lab Server. The recipient of a Web Service call transmitted over SSL can always identify the origin of the call. We use this mechanism to allow the Lab server to generate a two part unique key to identify experiments: (1) the **`experimentID`** unique on a particular Service Broker and (2) the Service Broker's identity, which can be determined from each SSL-transmitted web service call. It initially would seem more natural to have the Lab Server generate the unique **`experimentID`**, but there is a reciprocal problem because Service Brokers may submit experiments to multiple Lab Servers. Clearly, Service Brokers must not submit two experiments with the same **`experimentID`**.

Service Broker and Lab Server Communication

The Service Broker and Lab Server communicate via the web service calls described in the following section. Individual users are authenticated by the Service Broker, which vouches for them by forwarding their experiment specifications on to the Lab Server and then retrieving the results. In order to protect the Lab Server from unauthorized and potentially mischievous use, the Service Broker must authenticate itself to the Lab Server. The Lab Server can invoke only one method on the Service Broker, the **`Notify()`** call. Although the danger of unauthorized access of a Service Broker by a rogue or counterfeit Lab Server is much less than the reverse case, we have specified a symmetric procedure by which the Lab Server can authenticate itself to the Service Broker. On each web service call, the invoking server identifies and authenticates itself to the target of the call by passing a unique string **`ID`** and passkey in the SOAP header of the web service call. The current implementation uses SSL to transport the web service SOAP messages so the ID and passkey are encrypted and secure from interception. If we find that the overhead for SSL is too high, we will need to change the authentication mechanism to keep it secure. Because the ID and passkey only appear in the header, they do not appear in the argument lists of the methods below. The administrative methods required on the Service Broker to manage these IDs and passkey are described in the *Service Broker Administrative API*.

Service Broker Preparation

For the following, please refer to on P. 6.

1. The Service Broker creates and stores (using the **`GenerateServerID()`** method of the *Service Broker Administrative API*) a unique GUID-based ID that it will use to identify itself to all Lab Servers.
2. This ID is communicated out of band to the administrator of a Lab Server to which the Service Broker will submit experiments. The administrator of the Lab Server generates a similar GUID-

based ID for the Lab Server if one has not already been generated. He also generates a unique secret passkey that the Service Broker will use to authenticate itself to the Lab Server. The pair of Service Broker ID and passkey are stored on the Lab Server to be used for the future authentication of web service calls.

3. The Service Broker passkey and the Lab Server ID are conveyed out of band back to the administrator of the Service Broker. She stores the pair of the Lab Server ID and the passkey the Service Broker is to use to contact that Lab Server using the **`RegisterOutgoingServerPasskey()`** method.
4. The Service Broker administrator generates and stores a unique secret passkey that the Lab Server will use to authenticate itself to the Service Broker using the **`GenerateIncomingServerPasskey()`** method. She conveys this passkey out of band to the administrator of the Lab Server who stores it on the Lab Server paired with the Service Broker ID. The procedures on the two systems are symmetrical, but the implementation on the Lab Server need not follow the same API as that on the Service Broker.

Figure

Executing an Experiment

For the following, please refer to on page 7.

1. A user authenticates herself to the Service Broker and selects a Lab Client to use for the duration of the current session.
2. After she has prepared an experiment specification using the Lab Client, she submits it to the Service Broker using the WS method **`Submit()`** that is part of the *Client To Service Broker API*.
3. The Service Broker establishes an SSL connection to the Lab Server for which the experiment is targeted.
4. The Service Broker invokes the WS method **`Submit()`**from this API on the Lab Server over the SSL connection. The header of the SOAP request contains the Service Broker GUID-based ID (returned by **`GetServerID()`**) and the secret passkey assigned to the Service Broker by the Lab Server on which the method is

invoked (returned by **`GetOutgoingServerPasskey()`**). The method returns with a **`SubmissionReport`**.

5. When the Lab Server has completed execution of the experiment, it invokes the WS method **`Notify()`** from this API on the Service Broker. The header of the SOAP request contains the Lab Server GUID-based ID and the secret passkey assigned to the Lab Server by the Service Broker on which the method is invoked. The passkey is validated by the copy the Service Broker retrieves using the method **`GetIncomingServerPasskey()`**.
6. The Service Broker retrieves the experiment results in the form of a **`ResultReport`** object by invoking the WS method **`RetrieveResult()`**from this API on the Lab Server over the SSL connection. The header of the SOAP request once again contains the Service Broker GUID-based ID and the secret passkey.

Figure

Methods

Service Calls from Service Broker to Lab Server

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 identifying the experiment returned from a previous call to **`Submit()`**. */

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 userGroup`
/* Effective group of the user submitting the hypothetical new experiment. */

`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. 9. */

`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 userGroup`

/* Effective group of the user requesting the lab configuration. */

Returns:

`string labConfiguration`
/* An opaque, domain-dependent lab configuration. */

GetLabInfo

Purpose:

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

Arguments:

none

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:

none

Returns:

`LabStatus theStatus`
/* 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 **`labExperimentID.`** */

`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:

`int experimentID`
/* The identifying token that can be used to inquire about the status of this experiment and to retrieve the results when ready. */

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

`string userGroup`
/* Effective group of the user submitting this experiment.
*/

`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. */

Returns:

`SubmissionReport submissionReport`
/* See Description Below. */

`public struct submissionReport`

{

`public ValidationReport vReport`

/* See description below, under . */

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

`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. 9. */

}

Validate

Purpose:

/* Checks whether an experiment specification would be accepted if submitted for execution. */

Arguments:

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

`string userGroup`
/* Effective group of the user submitting this experiment. */

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-dependent 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. */

}

Service Calls from Lab Server to Service Broker

Notify

Purpose:

/* Notifies the Service Broker that a previously submitted experiment has terminated. */

Arguments:

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

Returns:

`Void none`