The iLab Service Broker: a Software Infrastructure Providing Common Services in Support of Internet Accessible Laboratories

by

Karim Y. Yehia

B.Eng. Mechanical Engineering
American University of Beirut, 2002

SUBMITTED TO THE DEPARTMENT OF CIVIL AND ENVIRONMENTAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUNE 2004

Signature of Author: ....[illegible]....................
Department of Civil and Environmental Engineering
May 13, 2004

Certified by: ....[illegible]....................
Steven R. Lerman
Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by: ....
Heidi M. Nepf
Chairman, Committee for Graduate Students

BARKER

The iLab Service Broker: a Software Infrastructure Providing Common Services in Support of Internet Accessible Laboratories

by

Karim Y. Yehia

SUBMITTED TO THE DEPARTMENT OF CIVIL AND ENVIRONMENTAL ENGINEERING ON
MAY 13, 2004
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

ABSTRACT

Remote laboratories that are accessible via the Internet are becoming a common phenomenon in higher education institutions. This thesis describes the iLab Service Broker architecture, a software infrastructure that supports these Internet-accessible labs by providing a number of services that facilitates their administration and management. These common services are authentication, authorization, lab administration, scheduling and data storage. While Internet-accessible laboratories may be quite varied in terms of the technologies they use, they tend to have similar topologies, consisting of a lab client, lab server and database. The end-user interacts with the lab client to issue commands to, and view results from the lab server, and the database stores data from the lab server. The Service Broker's internal architecture constitutes the business logic rules that govern how the common services are administered. Its external architecture exposes these services, using web services, and makes them available to the remote laboratories in a platform-independent manner. The Service Broker attempts to cater for the different experiment models that Internet-accessible labs host. These models are the batched experiment, the interactive experiment and the sensor lab experiment.

Thesis Supervisor: Steven R. Lerman
Title: Professor of Civil and Environmental Engineering

ACKNOWLEDGEMENTS

I would first like to thank my thesis supervisor, Prof. Steven R. Lerman for giving me the opportunity to work on the iLab project. Prof. Lerman has always been a source of guidance and motivation for my work in this project and at MIT, and our research conversations always enlightened me.

I would also like to extend my gratitude to Dr. V. Judson Harward, the iLab project's principal investigator. Judson continues to do an excellent job at leading the team's research meetings, pouring in creative ideas, and arranging for relevant parties to become involved in the project. I never cease to be amazed at such dedication for this project.

I would also like to thank my parents Lina Yehia and Yehia Yehia for their continuous support of my decisions and their belief in me. It is because of their efforts that I have made it this far. My sisters and best-friends, Nayla Yehia and Yasmina Yehia, have given me all the encouragement and advice anyone could ask for. My extended family and friends, Raja Hassen, Jawad Yehia, Azza Yehia, Wafa Halawi, Zeina Fakhry, Leila Hassen, Ali Hassen, Nada Hassen, Fadi Husseini, Hussein Dajani, Leila Hoballah, Fadi Abu Ibrahim, Loai Na'amani, Ralph Rabbat, and Bassel Younan have supported me in my endeavors and have made my university experiences forever memorable. My sincerest appreciation goes out to all of them.

I would also like to thank Charuleka Varadharajan, Shaomin Wang and Jedidiah Northridge, my team members in the iLab project, for their cooperation and valuable advice. Their commitment and contribution to this project will always be an inspiration for me.

Table of Contents

Table of Figures

Chapter 1. Introduction

The past few years have witnessed a significant integration of web and Internet technologies in higher education. This trend has brought about radical pedagogical transformations, and has sparked some exceptional initiatives that have made educational resources, which were once confined within university boundaries, more accessible.

Many academic researchers have taken advantage of connectivity across the Internet to make their laboratory facilities Internet-accessible and provide their students with a hands-on experience outside of the lab. We have seen this phenomenon at MIT and many other institutions, for example, refer to [1], [2], and [3]. These Internet-accessible labs have made it possible to transcend space and time as they can now be operated anywhere and at anytime.

With this capability however, the lab owner or operator is faced with a host of administrative responsibilities: Distant users have to be authenticated, lab resources have to be properly managed and allocated, and student data needs to be stored somewhere, to mention a few. Since the actual lab facility and operation are their main focal points, many lab owners do not have the time or even wish to devise an extensive administrative / management system.

Additionally, as these Internet-accessible labs mushroom across various academic institutions and become a core part of many student curricula, the lab end-user will need to maintain a separate set of user information for each Internet lab account. This can become especially burdensome if the user is registered with multiple labs.

These points suggest that there is a need for a "centralized" entity that provides a set of services to leverage the common tasks required by these Internet-accessible labs. This entity would also serve as an entry-point and management system for the labs an end-user is subscribed to, much like a web-based course management system would for the courses a student is registered in. Internet-accessible labs are quite varied and tend to be constructed in ad hoc ways that are well suited to a particular lab's needs, and hence, this entity would need to provide its services independently of any particular platform. Additionally, the services need to be exposed in a manner such that minimal integration time is required by the lab owner or operator who uses them.

The following paragraphs describe Internet-accessible labs in general and the core ideas surrounding this "centralized entity" which we have come to call the iLab Service Broker.

1.1. A New Vision for Internet-Accessible Labs

A number of names have been used to refer to "Internet-accessible labs" – virtual labs, remote labs, web-enabled labs. These labs have been around for a while. By 1996, a network application called "Second Best to Being There" (SBBT) was developed at Oregon State University that allowed students to conduct control experiments remotely via the Internet [1].

Many Internet-accessible lab developers claim that their main motive is to re-create the laboratory experience as closely as possible, and seek to do so by providing graphical user interfaces (GUI), streaming video and sound, and collaboration tools. Internet-accessible labs have had, and will continue to have, important implications in web-based distance education. In her paper on web-based course management systems, Dabbagh describes a "radical constructivist" approach to web-based education tools in which students assimilate their knowledge base by contributing to and interacting with their learning environments [4]. Internet-accessible labs present scenarios where the student is a key participant in the learning process, and encourage "active learning". This is opposed to the more traditional instructional methods that consist of imparting educational materials to students "out of context". Active learning assumes that knowledge transfer is difficult and is facilitated by context learning [5].

The introduction of Internet-accessible labs has also provided academia with a considerable "reach" advantage. Students at one institution can perform resource-intensive experiments at other institutions at a much lower cost than that associated with acquiring the actual lab facilities. These facilities may have a high capital cost, in addition to specialized maintenance requirements. Moreover, these remote labs allow geographically dispersed students to collaborate on an experiment and bring forth an amalgamation of research ideas, for example, refer to [6].

1.1.1. The Anatomy of an Internet-Accessible Lab

A number of Internet-accessible lab initiatives have been deployed at MIT in different departments. Some of these are the *MIT Microelectronics WebLab*, the *Remote Polymer Crystallization Experiment* lab, and the *MIT Flagpole* project. These three labs actually represent different experiment paradigms; this point will be revisited in a subsequent section.

What these labs do share in common is an architecture whose components can be broken down into what are effectively a lab server, a lab client and a backend database. Typically, lab users will interact with a lab client, which often comes in the form of a GUI that accepts input parameters to be transmitted to the lab apparatus, and displays experiment results in one form or another. The lab server represents the software interface that converts messages from the lab client into a form that can be understood by, and used to control the lab hardware. The database may be used for a number of purposes, from storing experimental results, to storing user data to allow for persistence across multiple web sessions.

The way that these lab components (lab client, server and database) communicate in executing an experiment also has some common features. An experiment specification made on the lab client should be *validated* before it is transmitted to the lab server, otherwise it may potentially damage the lab apparatus. A valid experiment specification should then be *submitted* to or the lab server. Once the results are ready, they should be *retrieved* and relayed back to the lab client. The keywords here have been italicized, and it is these common points of functionality (and a few others) that represent one premise upon which the iLab Service Broker has been built.

1.1.2. Separation of Domain-Independent and Domain-Specific Aspects

Another premise that guided our design of the Service Broker was the abstraction of those tasks that were common to or desired by most labs. We have termed these "domain-independent" as they are not associated with any particular type of lab. Essentially, we have identified five high-level mechanisms that most Internet-accessible labs may want to implement:

1. An experiment-storage mechanism to store all specifications pertaining to an experiment type or run, and all results returned by an experiment.
2. An authentication / security mechanism to establish the identity of the user and setup a secure web-connection with which to communicate with the remote lab.
3. An authorization mechanism to specify each remote lab user's privileges on the lab server and database.
4. A reservation mechanism to allocate time slots for experiments.
5. An administrative mechanism to manage user subscriptions, accounts, and memberships in groups.

We do not advocate that remote labs implement all these mechanisms, and they may actually be irrelevant for certain types of labs. However we try to be comprehensive in providing domain-independent functions that many labs will want to use.

We also have domain-specific aspects of Internet-accessible labs that we do not consider in our design of the Service Broker. One such example is how a floating-number value representation of a particular voltage is translated into a bunch of binary digits, which the lab hardware understands. It is sometimes difficult to draw a clear distinction between domain-independent and domain-specific functions. Generally speaking, those aspects that are hardware or vendor specific or just too fine-grained are not considered to be common Internet-accessible lab tasks.

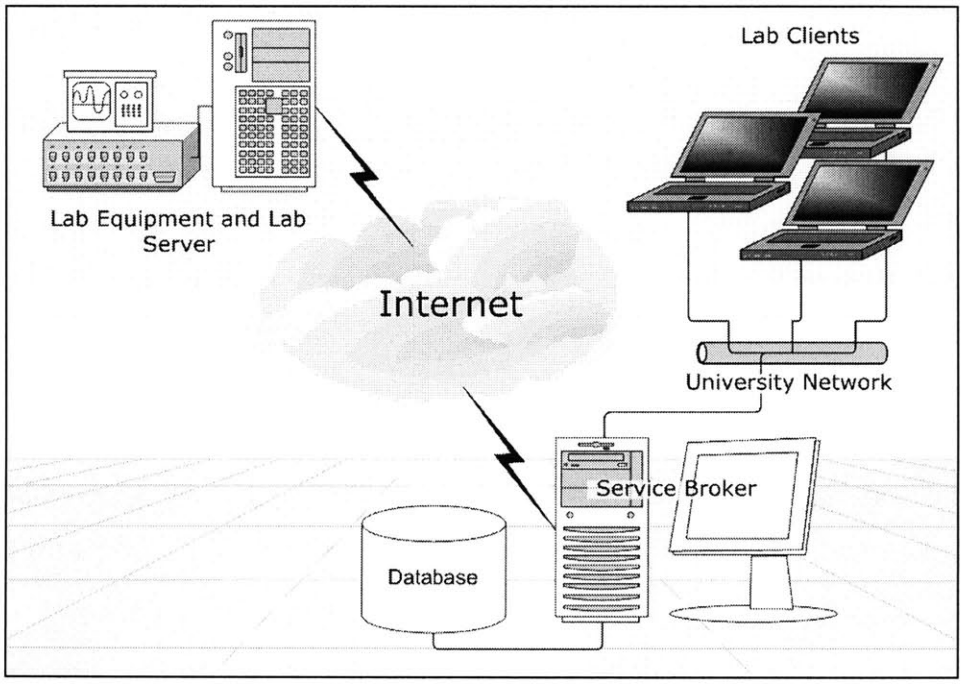


Figure 1. A Generalized iLab architecture showing the lab client, lab server and Service Broker components. Figure adapted from [7].

1.1.3. The iLab Service Broker: An Intermediate Component

The arguments presented thus far document the need of an infrastructure that can perform two central operations. These are (1) to facilitate the tasks that are common to Internet-accessible labs, and (2) allow for the exchange of messages between this infrastructure and the lab components. The former point refers to the Service Broker's business logic (the internal architecture), while the latter refers to its exposed interface (the external architecture).

The Service Broker can be regarded, along with the lab client and server, as another component within a more "generalized iLab architecture". This is illustrated in Figure 1.

Although vendors like National Instruments and Agilent provide tools that help make labs web-enabled, there is no 'magic recipe' to create an Internet-accessible lab. Internet labs are extremely varied in terms of the platform they are developed on, and the technologies they use. One critical requirement for the external Service Broker architecture is that it allows for the interoperability of Internet lab components regardless of the platform they are operating on or the language they are written in. The words "interoperability", "component" and "broker" seem to suggest there is some reference here to the Object Request Broker (ORB) architecture. ORB is a middleware component that manages requests between clients and a CORBA object, which represents a real-world object and consists of data and methods that may be invoked on it [8]. Solutions built on CORBA, however, will be dependent on the vendor or developer's implementation and therefore do not achieve complete interoperability [9]. As such, they are not well suited for an iLab Service Broker implementation.

We have found that web services actually provide the platform-transparency that the Service Broker's exposed interface needs. This is because the web service technology stack is comprised of widely accepted and open specifications and Internet standards like HTTP, XML, WSDL and SOAP [9]. As a result, web services permit client and server applications to be loosely coupled and therefore achieve portability. Web services are often described as SOAP over HTTP. SOAP (Simple Object Access Protocol) is a lightweight, object-oriented protocol based on XML, which is a cross-platform standard for formatting and organizing information [10].

As a general design principle, the Service Broker services have been developed in a policy-neutral fashion. That is, by integrating with the Service Broker, Internet labs are not required to enforce any *particular* policy on their clients. We try to provide the services in a form that allows the developers to customize their own policies.

1.2. Taxonomy of Internet Lab Experiments

After carrying out a survey of existing Internet-accessible labs, we have been able to distinguish three different categories of Internet lab experiments.

The first of these, the "batched experiment" paradigm, appears to be the simplest. A batched experiment refers to one in which an experiment specification is made and submitted. The experiment then proceeds without any further lab user intervention. Once the experiment terminates, the results may be collected and reviewed. The *MIT Microelectronics WebLab* is an example of such a lab. Here, the lab client is a Java applet and it presents its users with an array of microelectronic devices, such as transistors and diodes, and users are expected to provide a range of voltages or currents, which are input to these devices via the lab server. The lab server will correspondingly vary the voltage or current level on these devices, and collect measurements for each variation. At the end of the experiment, a graphing utility displays the results to the clients, which may also be downloaded. Typical characteristics of such an experiment type include short running times, and that they are often unaffected by network latency.

The second type of experiment, the "interactive experiment" refers to a case where messages are, more or less, continuously exchanged between the lab client and server. That is, the lab user makes experiment specifications on an ongoing basis as results are displayed on the client. At the end of the experiment, the lab user will have accumulated a repertoire of results that can be used to replay the entire experiment. The Department of Chemistry at MIT has a *Remote Polymer Crystallization Experiment* lab, which is interactive. Here, the user controls the heating of a polymer sample on a movable stage. The sample, as it cools, gradually forms a number of crystal nuclei that are observed through a polarized light microscope. Photographs of the sample are taken at various intervals to capture the formation and growth of the polymer crystals.

After the experiment has terminated, the lab user may analyze the crystal images. Compared to the batched experiment model this one takes a longer time to complete, requires more user interaction and performance may be degraded by network latency.

The final experiment paradigm is that of the sensor experiment, which involves the real-time streaming of sensor data to the lab user with minimal user intervention. The streaming data may be delivered in a raw or processed form. A user could possibly interact with a sensor by subscribing to a trigger or an alert, which is a particular event in the streaming data that the user is interested in. When the trigger condition is met the user is notified, possibly by email. The *MIT Flagpole Project* hosted in the Department of Civil Engineering is one example of a sensor experiment. This project consists of a number of accelerometers attached to a flagpole in an MIT courtyard. The accelerometers stream displacement, stress and strain data that are broadcast through the Internet and can be viewed with an applet client. The sensor experiment model is complicated by the actual nature of sensor data streams: Data is produced continuously without having been explicitly asked for that data. Often, the data needs to be processed in real-time because it may be expensive to store it in a raw format and because of the need to understand the real-world events that the data represents. Additionally, sensors have a limited power supply and their streams may be corrupted by noise. In an attempt to address some of these issues, Madden and Franklin have proposed an architecture for querying streaming data in [11]. Sensor experiments will be further explored in Chapter 7, Future Directions and Limitations.

One may wonder whether these experiment types can be placed under the umbrella of a single iLab Service Broker architecture, as they each display a different set of characteristics. It was not immediately apparent to us but an architecture that is well suited to the interactive experiment could be made to cater for the sensor and batched experiments, whereas the converse is not necessarily true. This idea will be expanded in Chapters 6.

1.3. Development Environment and Methodology

The iLab Service Broker project is part of the iCampus initiative, which is a MIT-Microsoft alliance geared towards using information technology to enhance and improve university education. The Internet-accessible labs mentioned in the previous section are also a part of iCampus, and have served as an on-going reference for our design decisions. At the same time, our architecture does not lose sight of other remote labs out there and is developed in the more general context.

We have designed a number of application programming interfaces (APIs) that correspond to the Service Broker business logic and exposed interface. We have also drawn up some data models to be used in the backend database. These have been implemented and successfully tested in a first-round prototype (version 3.0.0.3); The *MIT Microelectronics WebLab* lab fully integrated our services and was used in an electrical engineering course consisting of about 100 students. The overall architecture proved to be robust. Screenshots of the iLab Service Broker and WebLab GUIs are shown in Appendix A. The current iteration includes a redesign of APIs and data models to better accommodate interactive experiments.

For our initial iteration we worked closely with the WebLab team and prioritized their requirements from our Service Broker architecture. The point of this was to establish the functionality of our API's core methods, and proceed on a course that would reveal what aspects of our architecture needed to be revisited. This piecewise design methodology has proved to be effective, as it did not undermine our ability to keep the problem at hand within a manageable and understandable scope.

1.4. Thesis Roadmap

The remaining sections of this thesis are structured as follows: Chapter 2 discusses how communication between the Internet lab components and Service Broker occurs. The methods in the external architecture APIs are presented, along with a description of how they have evolved since the first iteration.

Chapter 3 provides details on the core Service Broker business logic APIs, and Chapter 4 gives a more in-depth discussion of the iLab Authorization model and how it was implemented in the first iteration. Chapter 5 introduces the data models that correspond to the internal architecture APIs. Chapter 6 describes the Remote Polymer Crystallization Lab in more detail and how we have remodeled our architecture to leverage this interactive lab. Finally, Chapter 7 gives an idea of the direction this project is heading and what some of its current limitations are.

The ideas and concepts presented in chapters 1 through 5 are based on the collective efforts of the iLab architecture team at MIT. The team members are V. Judson Harward, Jesús A. Del Alamo, Vijay S. Choudhary, James L. Hardison, Steven R. Lerman, Jedidiah Northridge, Charuleka Varadharajan, Shaomin Wang, David Zych, and the author.

The author's main contributions towards original work and implementation include Chapter 4, the iLab Authorization Model, and Chapter 6, the iLab Interactive Experiment, to which Jedidiah Northridge is also a main contributor.

Chapter 2. The External Architecture

The external architecture, or equivalently, the exposed interface refers to the set of APIs that allows the iLab Service Broker to communicate with the other iLab components (the lab client and lab server). This interface is "exposed" because it must be implemented by Internet-accessible labs that want to make use of the iLab architecture and Service Broker services. Furthermore, it represents the boundary through which information flows to and from the internal Service Broker architecture where the business logic is implemented.

The external architecture designed in the first development cycle mainly catered to the batched experiment mode. This was a proof-of-concept design that leveraged the WebLab experiment and served to illustrate how practical and applicable our architecture was to Internet-accessible laboratories in general. The discussion below will show that the Service Broker plays an intimate role in this prototype since it takes part in *all* message exchanges between the iLab components. Chapter 6 will explain why this model is not necessarily applicable to interactive experiments.

Before our discussion of the external architecture, we first describe the programming environment the iLab prototypes are developed in and the technologies used in implementation. Our APIs were implemented in C#, a new object-oriented and Internet-centric language that shares some features of Java, Visual Basic and C++. C# was introduced as part of Microsoft's .NET platform, a framework that provides a new API for the Windows operating system. The .NET platform combines a number of existing technologies such as COM+ component services, and the ASP web development framework, as well as a commitment to XML and object-oriented design, and support for web services protocols such as SOAP, WSDL and UDDI. The Service Broker services were exposed as ASP.NET web forms and web services. ASP.NET builds on classic ASP, and makes it easier and faster to build dynamic and data-driven web applications. The Visual Studio .NET integrated development environment (IDE) was used to author the APIs and web forms. This IDE offers a number of tools that help expedite the development process, such as visual development of web pages, IntelliSense code completion and integrated debugging [10]. Microsoft's SQL Server 2000, a relational database management system (RDBMS) was used in our implementation of the backend database. The initial prototype was deployed on a Dell server machine (Intel Xeon CPU 1.80 GHz; 1 GB RAM) running a Windows Server 2000 operating system.

2.1. Service Broker Communication: Top Looking Down

We must make a distinction between the types of communication that take place within the iLab architecture: Intra-iLab communication refers to the exchange of messages between the various iLab components whereas Inter-institutional communication refers to the exchange of messages involving different institutions to allow for sharing of iLab resources on a more global scale.

2.1.1. Intra-iLab communication

The current prototype allows messages to be exchanged only between lab client and Service Broker, or lab server and Service Broker. There is no direct communication between the lab client and lab server components. We provide this layer of indirection because Lab server developers *often* author the corresponding lab clients and the two components *usually* use complementary technologies. Our architecture however, allows a third party developer, who need not be directly associated with the lab server, to construct a client based on some technology of their preference. This is possible because our layer of indirection allows lab components to communicate in a platform-independent way. Additionally, because all messages are routed through the Service Broker, security can be regulated and enforced in one central location.

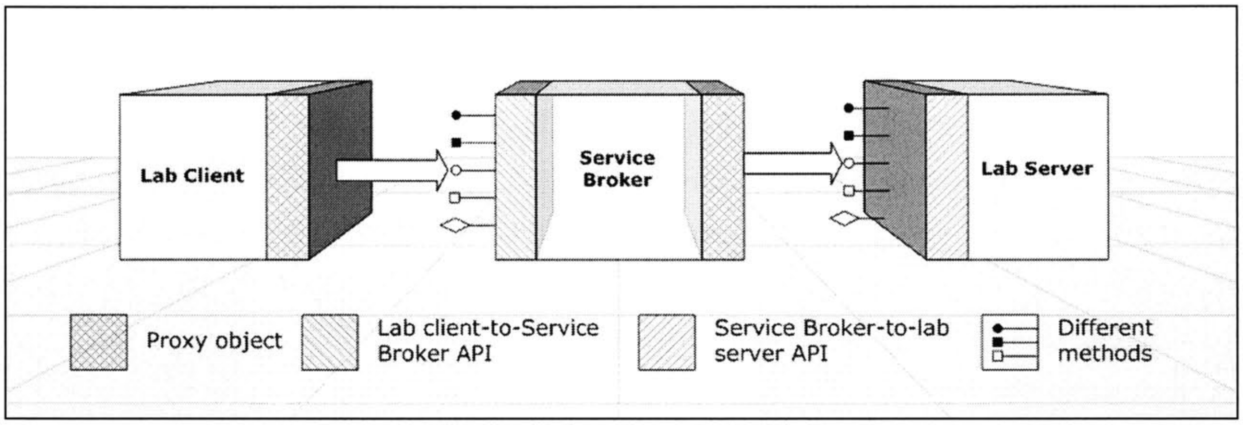

Lab Client
Service Broker
Lab Server
Proxy object
Lab client-to-Service Broker API
Service Broker-to-lab server API
Different methods

Figure 2. Figure showing where intra-iLab APIs and proxy objects reside. Arrows point in the direction of the interface whose methods are invoked

In order to enable intra-iLab communication, two APIs are exposed. They are the client-to-Service Broker API, which resides on the Service Broker and the Service Broker-to-lab server API, which resides on the lab server. These APIs have been exposed as .NET web services and consist of *pass-through methods*, that is, a method invocation in the former API will trigger the invocation of a corresponding method in the latter API and allow for a flow of information through the Service Broker business logic. A proxy object exists on the lab client to invoke methods of the client-to-Service Broker API, and another proxy object lives on the Service Broker to complete the method invocation of the Service Broker-to-lab server API. This is illustrated in Figure 2. The specifics of these APIs are detailed in sections 2.2.1 and 2.2.2.

Since the Service Broker vouches for all lab clients and transmits all of their messages to the lab server, the lab server must first authenticate the Service Broker. This trust is negotiated upfront before any messages are transmitted, with the Service Broker and lab server exchanging unique identifiers and passkeys. Thereafter, all API method calls involve the passing around of these parameters. This security mechanism is further discussed in section 2.2.1, the Service Broker-to lab server API.

The interjection of the Service Broker component does not come without a cost, and this is mainly due to the use of web services. One performance hit in web services is their underlying messaging and transport protocols. A request to a SOAP-based web service method begins by using the Web Services Description Language (WSDL) document to understand the nature of the API methods and parameters. Also, data is marshaled into XML SOAP requests and response documents to be moved across software packages using HTTP. These processes require some XML parsing and validation and are therefore time-consuming [12]. A batched experiment such as WebLab can tolerate this latency, but an interactive experiment which requires near real-time interaction cannot.

2.1.2. Inter-Institutional Communication

One ultimate goal of the iLab project is to establish the ubiquity of Internet-accessible labs. In an academic setting, this amounts to placing a Service Broker component in each institution that wants to grant its students the permission to use Internet-accessible labs in other universities, as well as its own. A student's identity is established within a particular Service Broker context. To access a lab server, the student's Service Broker must vouch for his or her identity, and the lab server must trust the Service Broker. Figure 3 shows how this inter-institutional communication happens. Here, students of institution A and can run experiments on Lab Server B, which is not aware of any specific user, but rather is aware of the Service Broker that has authenticated that user, which in this case is Service Broker A.

Lab servers will find this scheme elegant because it frees them from the burden of administering students – they can rest assured that they are dealing with trustworthy Service Brokers and not just any lab client.

One may wonder how such a scheme would allow for the fair / policy-based allocation of timeslots on an interactive lab that shares its resources among multiple Service Brokers. Although Service Brokers are not aware of each other's end-users, it is still possible to allocate time fairly among them since we envision a scheduling / reservation service that negotiates time allocations between one lab server and many Service Brokers.

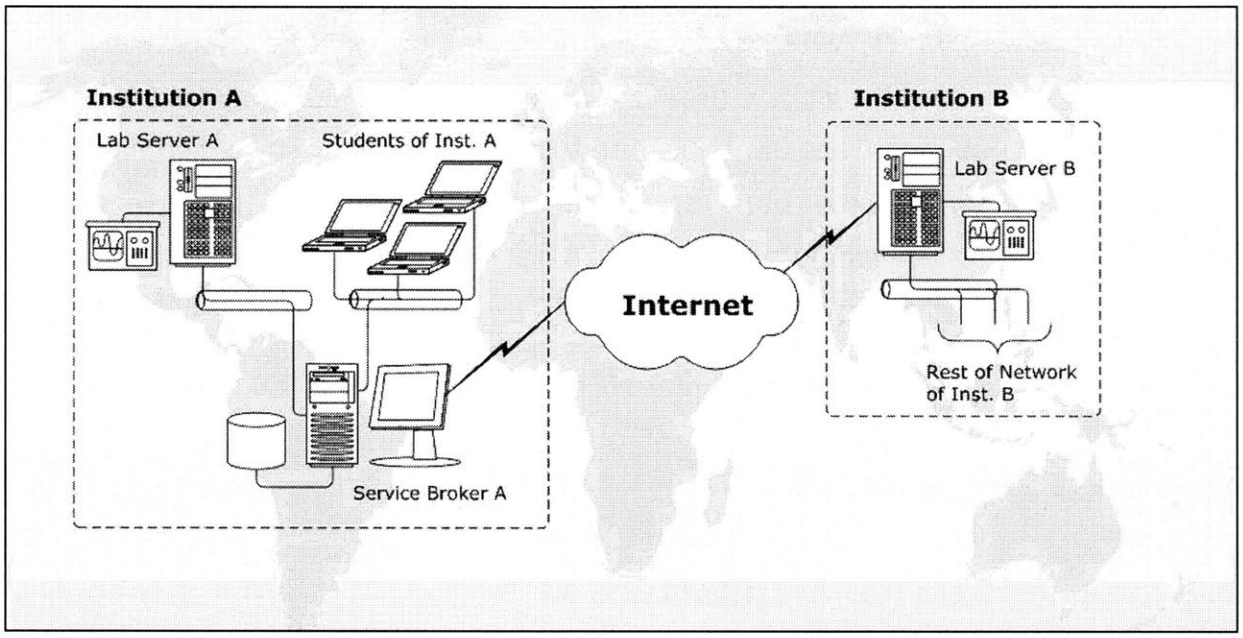


Figure 3. Inter-institutional communication: Student of institution A can gain access to Lab Server B via Service Broker A

This inter-institutional communication can be realized once the iLab Service Broker architecture is released in a shippable / configurable form and distributed among multiple institutions.

2.1.3. A Web Service Implementation

To recap from Chapter 1, web services were chosen to implement the external architecture because they are ideal for heterogeneous environments. That is, clients and servers can be running different architectures and platforms, yet are still able to communicate with each other using web services. An XML web service is essentially a remote procedural call (RPC) service in which requests and responses are encoded in XML as SOAP envelopes, and transported over HTTP [13].

By committing ourselves to web services, we have a clear interoperability advantage. However, we do relinquish some of the advantages of other proprietary RPC or messaging middleware. Typically, RPC technologies such as the Java Remote Method Invocation (RMI) or .NET Remoting protocols allow data to be transferred over the wire in some binary format, which is less bulky and faster to transport than XML. Web services can only be accessed at the application (HTTP) layer, whereas some proprietary messaging middleware can be accessed at the transport (TCP), in addition to the application layer. Moreover, web services are stateless and preservation of state requires the passing around of session objects, which is not necessary in certain RPC implementations where singleton objects can be used to maintain state [14].

For the purposes of the Service Broker, the interoperability advantage gained with a web service implementation outweighs its disadvantages. Additionally, because web services use XML and HTTP, they can penetrate firewalls and other restrictions.

2.2. The External Architecture APIs

The Service Broker communication described above can be accomplished using the Service Broker-to-lab server API and the lab client-to-Service Broker API. The currently deployed version of the APIs contain a number of methods that (1) allow the lab client and Service Broker to learn about the present state of the lab server and (2) allow the lab server to accept and validate experiment specifications, and to subsequently return experimental results. In the case of interactive experiments, Chapter 6 will explain why the validation of experiment specifications should not be routed through the Service Broker.

The APIs define a number of object types that represent real-life entities like lab server, user, and group. The Service Broker is the central "management" hub which stores these objects and can modify them. Often, copies of these objects or IDs that refer to them are passed to the other iLab components (the lab server and lab client) for various reasons. We wanted to emphasize that any changes made to the copy would not be reflected in the original object that lives on the Service Broker. We therefore defined these objects as "structs" which are value-types, rather than "classes" which are reference-types. Additionally, various immutable fields of these types such as object IDs were declared as "read-only" such that they could not be changed once set.

2.2.1. Service Broker-to-lab server API

Prior to any message exchange between the Service Broker and lab server, the two components must first acknowledge each other and setup a secure mechanism with which they can communicate. Both components will generate their own unique identifiers (as global unique IDs or GUIDs), which they will use to identify themselves to each other. Each component will also generate a passkey for the other. These passkeys and GUIDs are relayed between the two out of band in an encrypted email or telephone call. Thereafter, the combination of GUID and passkey must be included in the SOAP headers of *all* method calls made between the two entities. Each entity will only admit messages that have a valid GUID / passkey combination. This, in addition to the fact that communication takes place along an SSL connection, makes our security mechanism a very stringent one. There is already some functionality built into the Service Broker (specifically in the internal architecture's Administrative API) to generate and register GUIDs and passkeys. The lab server, however, is free to choose its implementation to do this.

Having established valid GUIDs and passkeys, we move on to the API core functions that have been implemented in the current version. These functions concern the following areas:

1. Lab state:
    - Query the lab server status, configuration and information.
    - The "status" of a lab refers to whether it is currently online or down, whereas "configuration" refers to what the user can do given the current lab setup. This is dependent on his / her privileges. "Information" implies lab-specific information resources. "Configuration" and "information" are readable string types whereas "status" is a user-defined type.

2. Experiment:
    - Validate an experiment specification to ensure it won't damage the lab hardware.
    - Submit valid experiment specifications to the lab server for execution.
    - Query experiment status once a specification has been submitted (for example, whether it is running or has terminated) or cancel the experiment.
    - Obtain an estimate of the wait time until experiment completion based on the length of the experiment queue. When an experiment has terminated, the lab server notifies the Service Broker.
    - "Experiment specification" is a domain-dependent string representation. The processes of validating and submitting an experiment specification respectively return "validation" and "submission" reports, which are user-defined types. These reports are explained in further detail in section 3.5, the Experiment Storage API.
3. Experimental results: Retrieve experiment results once they are ready. Results are returned in the form of a result report, another user-defined type.

For a comprehensive listing of methods and structs in this API refer to [15].

2.2.2. Lab client-to-Service Broker API

In order for the lab client to communicate with the lab server, we expose the lab client-to-Service Broker API, which contains "mirror images" of the methods described in the previous section. A user can log into the Service Broker and launch a lab client, which provides an interface to call methods on the lab client-to-Service Broker API such as validating or submitting an experiment specification. Such method invocations will pass arguments specified by the user to the Service Broker, which in turn will call corresponding methods on the Service Broker-to-lab server API after appending its GUID and passkey in the method SOAP header. Aside from shielding the lab server from direct contact with lab clients, method calls are channeled via the Service Broker to gain access to the business logic. This allows us to determine whether the user has the privilege to call a method, say on a particular lab server with a particular configuration, and also to store results in the database.

Additionally, this API specifies a number of non-pass-through methods. Typically these methods deal with client or user specific items. An item is a name / value pair. For example, this capability allows a user to save an experiment specification without validating or submitting it until another session. Here, item name is "experiment specification" and item value is the actual specification.

The lab client-to-Service Broker specification given in [16] provides a detailed listing of methods and structs in this API.

Chapter 3. The Internal Architecture

The internal architecture refers to the set of APIs that constitute the Service Broker business logic. These APIs provide the domain-independent functionality that most laboratories need to implement in order to become securely accessible to users across the Internet. They are useful to lab owners who are often preoccupied with integrating various technologies to achieve Internet connectivity, and simply don't have the time or the expertise to design a system that encapsulates all aspects of user management.

In order to make the APIs as reusable as possible we set a design principle to try to keep them free of any particular policy. Instead, they are in a form that allows lab owners to configure their own policies. One such example is in the authentication API where labs have the choice to use the native Service Broker authentication mechanism provided, or implement an external authentication scheme such as Kerberos.

The APIs encompass different aspects of user management systems and have been broken down as follows: administrative, authentication, authorization, experiment storage, and reservation / scheduling. APIs pertaining to the first four domains have been built in the initial prototype. The scheduling functionality was not implemented because it wasn't relevant to the WebLab experiment, where experiments are submitted to a queue and processed in a first-in first-out

(FIFO) manner. The scheduling component, however, is important for labs whose resources need to be allocated among many users at different times.

In the current prototype the external architecture was exposed as a web service while the internal architecture was kept hidden from outside processes. Only Service Broker ASP.NET processes (the external architecture web service and the web application) can access and make use of the business logic APIs. One disadvantage of this design is that it is not modular – labs that choose to become part of the Service Broker architecture cannot, for example, implement an external experiment storage mechanism. This design restricts the iLab architecture's topology. Since many labs want more flexibility, future versions of the system will include stand-alone experiment storage and scheduling services.

3.1. The iLab Web Application and User Sessions

An understanding of how a typical user session proceeds will facilitate our discussion of the internal architecture APIs. Figure 4 presents a detailed map of the iLab Service Broker web application, and the functions exposed by each web page. The session begins with the user first logging in at an ASP.NET web page. Authenticated users are redirected to the "Effective Group" page where they choose a role they will assume for the remainder of the session. The concept of *effective user group* will be introduced in a later section, but for now we can think of this as a group with a specific set of privileges. Depending on the user's privileges, he / she may log in as a member of a lab user group or as a member of an administrator group.

Let's first follow the path a lab user would take. The first page the lab user would see is the "My Clients" page, which enumerates the lab clients associated with the chosen effective group. Here, a lab client may be launched using the "Launch Client" button and an experiment performed. Whereas the Service Broker web application consists of only ASP.NET web pages and forms, the lab client might be based on some other technology, such as a Java applet as in the case of the WebLab client (refer to screenshots in Appendix A). The client appears in front of the My Clients page, thereby indicating that the user is still in the same session. Even after the user has closed the lab client, he / she remains in session until the "Logout" button is clicked or the browser hosting the ASP.NET page is closed. The controls on the lab client will invoke methods in the external architecture lab client-to-Service Broker web service, as illustrated in Figure 2.

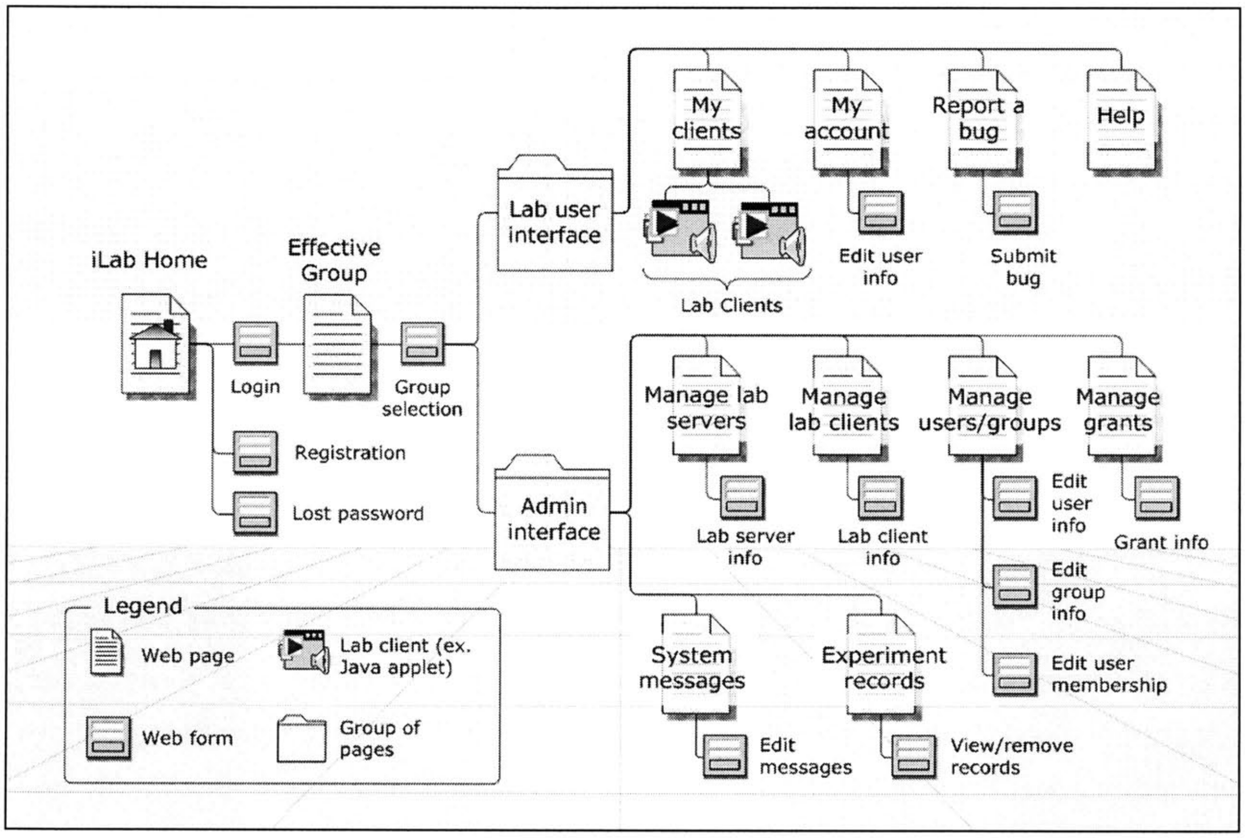


Figure 4. Figure showing the iLab web application structure and flow.

From the My Clients page, users can navigate to the "My Account" page where they can modify account information, reset their password, or request to join a new lab group. They can also navigate to the "Report a Bug" page or the "Help" page.

Moving on to the path an administrator would take, the user is presented with either all the pages shown in the administrator interface of Figure 4, or a subset of them as determined by the privileges of the group chosen in the Effective Group page. The current prototype has only a single administrator group called the "Super User" group, whose members have the authorization to access all administrative functions, including key tasks like managing lab servers and clients. Future versions will include administrator groups with more restricted access, such as a "Teacher Assistants" group whose members are authorized only to manage users and groups, and experiment records.

The "Manage Lab Servers" and "Manage Lab Clients" pages allow administrators to add, remove or edit existing lab servers and lab clients respectively. The "Manage Users / Groups" and "Manage Grants" pages permit administrators to establish authorization trees that will determine what each iLab user's privileges are. Section 3.3 describes some concepts behind the iLab authorization model and Chapter 4 discusses it in further detail. The "System Messages" page provides the capability to manipulate the messages displayed on certain ASP.NET pages and in the "Experiment Records" page one can review and annotate experiment records.

3.2. The Administrative API

The Administrative API provides much of the functionality exposed by the administrative interface web pages shown in Figure 4. This API defines several sets of methods, and each set acts upon one of the following value-types: User, Group, Lab Server, and Lab Client. Chapter 2 described that these objects were defined as structs, which are value-types, to emphasize the idea that any changes made to copies of them does not affect their original values.

The "User" type is special because different institutions might perceive it differently. Its definition contains a number of standard fields (such as user ID, first name, last name, etc.), in addition to a string field to hold extensible, XML-encoded information. For example, since "telephone number" is not a standard User field, a particular institution may choose to include this information in the XML-extension field. The contents of this field need to conform to a particular XML schema, which is set using one of the methods in this API. Another thing to note about this type is that, although a password needs to be associated with a user for authentication purposes, the password field has been abstracted away from the type definition. This is because "password" is a security-sensitive field and should not be sent on the network in the open. The association between user and password is maintained in the Service Broker database where passwords, as well as the other fields are stored.

Users and groups are collectively referred to as "agents". An "agent hierarchy" is a non-cyclical tree structure where the root nodes are groups and the leaf nodes are users. Intermediate nodes represent sub-groups. Sub-groups and users can have multiple parents. In the iLab architecture, the concept of "group" serves one primary purpose - it is a logical entity upon which authorization rules are applied.

The methods of the Administrative API that were implemented in the initial prototype allowed us to manipulate the aforementioned types as follows:

1. Lab servers: Add, remove or modify a lab server instance; Retrieve lab server instance(s) or lab server ID(s); Generate and retrieve passkeys and GUIDs which Service Brokers and lab servers use to authenticate themselves to each other.
2. Lab clients: Add, remove or modify a lab client instance (the add and modify methods allow us to associate a lab client with a particular lab server); Retrieve lab client instance(s) or lab client ID(s).
3. User: Add, remove or modify a user instance; Retrieve user instance(s) or user ID(s); Set XML schemas to validate the User XML extension field.
4. Group: Add, remove or modify a group instance; Retrieve group instance(s) or group ID(s); Retrieve information on group memberships and the agent hierarchy.

The specification given in [17] provides a comprehensive listing of structs and methods in this API.

3.3. The Authentication API

Authentication in the iLab project refers to the process of identifying clients that log into and use the resources of the Service Broker web application. Authentication happens across several tiers: The end-user's credentials are validated in the login page, requests made by valid end-users are checked for authenticity (one example of this is the GUID / passkey combination passed in SOAP headers of web service invocations), and connections to the database server are verified.

The Authentication API supports two types of authentication, native and external. What we have termed native authentication refers to a security mechanism that is only applicable to the iLab system. In contrast, external authentication refers to any enterprise-wide mechanism that is developed by a third party, and supports multiple systems of an organization. One such example is Kerberos at MIT. Provision for external authentication has been made in this API because many organizations are adopting a single authentication mechanism for the systems they use. Only the native approach has been implemented in the initial prototype.

Native Service Broker authentication is provided by ASP.NET "Forms authentication". This means that unauthenticated user requests are redirected to a login page (this has been labeled "iLab Home" in Figure 4), which contains a web form to collect the user's credentials. The form information is checked, and if it is valid the ASP.NET process attaches an encrypted session cookie that identifies the user for subsequent requests [18].

We use the term "principal" to refer to a valid client. Principal identities must flow through the web application to be used by the Authorization API to control access to resources on different tiers. This is accomplished by storing session parameters like user ID and the session group chosen by the user. These session parameters are passed to certain methods in the Authorization API at runtime to determine whether to grant access to certain resources. User-to-resource mappings are maintained in the backend database.

The Authentication API is a simple one and contains only a few methods that accomplish the following:

1. Manipulate native principal ID and password: Add and remove native principal IDs; Set or modify native passwords associated with a principal ID.

2. Check user credentials: The "Authenticate" method takes a single argument "Type" which could be either native or external. The user ID and password provided by the user are checked against the backend database.

Further details on these methods can be found in [19].

3.4. The Authorization API

The authorization mechanism in the iLab project is an elaborate one. This section is intended to introduce some of the high-level concepts behind this API, and details are left for discussion in Chapter 4.

Meier et al. describe two approaches to authorization in [18], a "role-based" one and a "resource-based" one. In the former approach users are divided among a fixed number of roles by the web application and each role possesses different privileges to perform operations on resources. Fixed identities (such as the web application's identity), which are *trusted* by resource managers, are used to access resources. The latter approach, on the other hand, requires that the user's original identity be conveyed at the operating system level and access to resources is determined by each resource's access control list (ACL)[1].

For the iLab Service Broker, we implemented the role-based approach because it presented several advantages over the resource-based one, the primary advantage being one of scalability. With this approach, our resource manager, the backend database server, needs to trust only a single fixed identity, that of the Service Broker application. This allows for effective database connection pooling. That is, our application can cache and reuse a database connection for many requests to access resources. In a resource-based mechanism, we wouldn't have been able to effectively use connection pooling since all database connections are tied to the individual security context of the

[1]An access control list (ACL) is a table that tells a computer operating system which access rights each user has to a particular system object, such as a file directory or individual file.

original callers. Logging into and connecting to a database are expensive operations [18].

Although a single fixed identity communicates with the resource manager, the identities of individual users must be conveyed at the application level. This is to allow per-user access to certain resources, such as an experiment performed by a specific user. The preceding section explained that user ID and effective user group were set as session parameters upon login. These parameters are then passed to stored procedures in the backend database to allow us to retrieve and process user-specific data. For example, suppose we have the following stored procedure:

```
SELECT * From stored_item_summary Where User_ID = @userID
```

having a single parameter, `@userID`. If "jsmith" is passed to this procedure, then the field "stored_item_summary" will be retrieved from a record that is associated with the user ID "jsmith".

Earlier on in the chapter, the concept of "effective user groups" was introduced as groups with different sets of privileges, and hence these groups can be regarded as having different roles. Figure 4 shows that an effective user group selection is made after login, and this role persists for an entire session until the user logs out. A single user can be a member of multiple effective user groups. This capability allows members of a course administrator group, for example, to assume the role of a course student in case they need to help a student solve a particular problem. However, users can only login into the web application with a single role in any particular session.

The iLab Service Broker authorization model has an internal system architecture that defines how authorization policy is enforced and this is consistent across all Service Broker architectures. The model also has an external social design that allows administrators to configure policies since these may differ from one campus to another. Both of these aspects are further explored in Chapter 4.

3.5. The Experiment Storage API

This API presents a number of methods that allows users to store, retrieve and delete experiment records. In Service Broker terminology, an "experiment record" is a log that documents various aspects of an experiment, such as messages exchanged between the lab server and lab client. Experiment records will vary from one lab to another, but for most *batched* experiments they would consist of three distinct parts: The lab server configuration at the time the experiment was carried out, an experiment specification submitted to the lab server, and the experiment results returned by the lab server. The Service Broker will not understand these experiment record parts, which are stored as strings, as they are domain dependent. However, since this API is intended to provide the functionality to search and manipulate experiment records we define a generic "experiment information" struct that contains fields that are understood by the Service Broker.

The experiment information struct allows administrative data like "user ID" and "lab server ID", and other experiment-specific data like "experiment submission time" and "experiment completion time" to be stored. This information is fixed for a particular experiment and, as such it cannot be changed once set. The struct does contain a modifiable string field, "annotation", which is a placeholder for a caption or a description that the user can set and modify at any time. Additionally, this type provides two extensible XML-encoded string fields, similar to the one present in the "User" type of the Administrative API. The first field pertains to experiment results and the second to binary large objects (BLOBs), which are explained below. The purpose of having these fields is to lay out experimental data in name-value pairs such that search rules can be applied to, and relevant information extracted from them. The fields must conform to a schema that is set by the lab server administrator using the methods in this API. The fields of the "experiment information" object simply facilitate the management of experiment records, and although they contain some administrative information, there is no referential integrity between Experiment Storage API objects and Administrative API objects. That is to say that if a particular user is deleted from the system, the experiment records that contain that user's ID are not deleted unless specified by some policy set by an administrator.

A binary large object (BLOB) is a large file, such as an image or sound file that must be handled (for example, uploaded, downloaded or stored in a database) in a special way because of its size [20]. BLOBs can be written to a database either as binary or character data. BLOBs can be stored in a database as a single value using the standard 'INSERT' or 'UPDATE' SQL commands, but if the object is very large, this may consume extensive system memory and reduce application performance. To prevent this, some implementations store BLOBs as "chunks" [21]. Because some labs will output experimental results in the form of BLOBs, the Experiment Storage API provides a number of methods for managing them. An association between a particular experiment record and BLOB is made using the "BLOB information" struct, which pairs every BLOB with an experiment.

The Service Broker-to-lab server API (section 2.2.1) defined a number of structs that can be manipulated by the methods in this API. "Validation report" is an object returned by the lab server upon validating an experiment specification. It consists of a number of fields that indicate whether the experiment specification can be submitted to the lab server for execution, and if not, what errors are associated with it. "Submission report" is returned when the experiment specification is submitted. It indicates whether the specification was accepted by the lab server, and includes a validation report as one of its members. It also indicates how long the associated experiment record will be stored on the lab server before it is purged. It is the responsibility of the Service Broker to retrieve and store the record before the lab server purges it. The last struct, "result report", is returned by the lab server when an experiment specification is submitted. This object maintains the status of the experiment, for example, whether the experiment is currently running or has terminated normally. It also holds a "lab configuration" string, and an "experiment results" string for when results are ready.

The methods that were implemented in the initial prototype provided the following experiment storage functionality:

1. Setting XML schemas to validate the extensible XML results and BLOB fields.
2. Manipulate experiment records: Create a new experiment record with a unique experiment ID; save or retrieve experiment specifications, reports, and lab configurations; remove experiment records from the database.
3. Manipulate BLOBs: Save, retrieve and remove BLOB objects associated with certain experiment records.

The specification given in [22] provides a comprehensive listing of methods and structs in this API. Chapter 6 will introduce a new experiment storage model, which has been broken away from the Service Broker, and whose methods are exposed as web services such they can be accessed directly by various iLab components.

3.6. Interactions between Different APIs

The internal APIs define a number of types that correspond to different iLab entities and components, for example, User, Lab Server, Lab Client, etc. One property of the APIs is that their methods are static and independent of these types, and are called upon an instance of the API class, which is part of a Service Broker namespace. By working within a Service Broker context, any change made to the state of a Service Broker is immediately reflected to all users working within the same context. For example, the modification of an existing lab server by an administrator becomes visible to all people logged onto the Service Broker. It is for this reason that the "Modify Lab Server" method is a member of the Service Broker's Administrative API class and not a member of the Lab Server type.

Figure 5 is a rendering of how the various APIs interact with each other. The arrows point in the direction of APIs from which methods are called. Notice how all the internal architecture APIs (Authentication, Authorization, Administrative, Experiment Storage and Scheduling) invoke methods on an "Internal Database" API. This API exposes methods that run various stored procedures on and extract information from the database management system (DBMS). One example is the "Authenticate" method in the Authentication API that checks whether the supplied user credentials match the ones in the database.

Also note that only authorized users can call the methods in the Administrative, Experiment Storage and Scheduling APIs, and hence calls must first traverse the Authorization API layer. Thus, calls that happen past this Authorization layer are executing in trusted code.

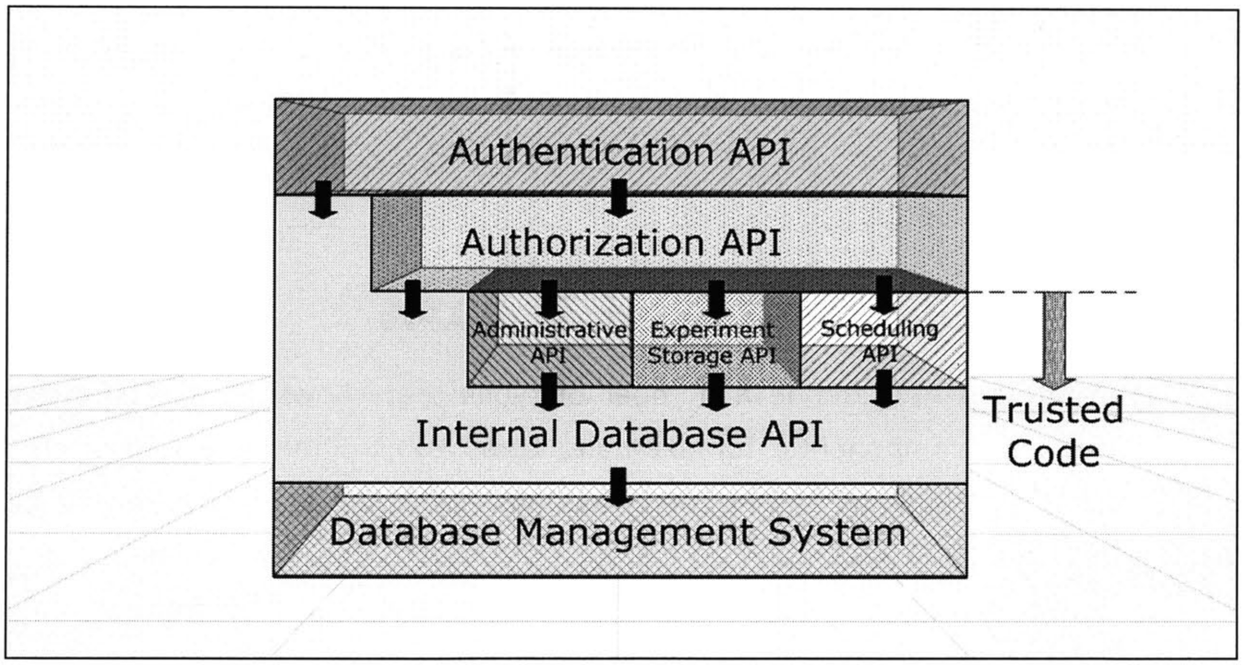


Figure 5. Figure showing how the Internal Architecture APIs are layered and which layer calls methods in other layers

Chapter 4. The iLab Authorization Model

The iLab Authorization model was briefly introduced in Chapter 3. It uses a role-based approach that assigns users logging into the Service Broker web application to different effective-user groups, with each group having a different set of privileges to perform operations on the Service Broker.

This model draws some concepts from the Open Knowledge Initiative (OKI) Authorization OSID. The OKI project presents an open and extensible architecture for general purpose infrastructure, and targets mainly higher education applications. OKI offers interface definitions, referred to as Open Service Interface Definitions (OSIDs), and each OSID defines an area of functionality. The OSIDs describe what is expected from a service, and implementation details on how the service is to be delivered are left to the developer to decide [23].

This chapter describes how the OKI Authorization definitions were adapted to and implemented in the iLab project. It also present a number of Authorization use-cases.

4.1. Grant: The Basic Authorization Unit

The privileges that a user has upon logging into the Service Broker web application are represented by instances of another user-defined type called

"Grant". Each Grant consists of three strings; an "agent", a "function" and a "qualifier". Semantically, this collection of strings can be thought of as a grammar where the agent is the subject, the function is the verb, and the qualifier is the object [24]. For example,

$$\underbrace{\text{User kyehia}}_{\textbf{Agent}}\ \underbrace{\text{uses the lab server}}_{\textbf{Function}}\ \underbrace{\text{"WebLab".}}_{\textbf{Qualifier}}$$

Programmatically, agent is a User or a Group struct, function is a member of an enumeration of operations that can be performed on the Service Broker, and qualifier is the entity upon which the operation is performed, and can be any one of a number of structs, such as User, Group, Lab Server, etc. In the above example the agent is a User object with ID "kyehia", and the qualifier is a Lab Server object with ID "WebLab".

These Grant objects allow us to construct very specific authorization rules, and this is attractive for lab owners because they can customize their authorization model in whatever way they want.

The term "agent" was first introduced in section 3.2, as a concept that refers to either a User or a Group object. There is no "agent struct" per se, and agent ID is simply a placeholder for a User or a Group object ID. Users and Groups form an implicit "Agent hierarchy", which is a graph structure having Group objects as root nodes and intermediate nodes, and User objects as leaf nodes.

Similarly, "qualifier" is a concept that refers to existing iLab structs. However, it is unlike "agent" in that the Authorization API actually defines a Qualifier struct. This type consists of three members; a string qualifier ID, a string qualifier reference ID, and an array of qualifier parents. The qualifier reference ID is the ID of the struct which the qualifier points to. In the above example, if "WebLab" is the Lab Server ID, it would also be the qualifier reference ID. The array of qualifier parents permits us to construct an explicit qualifier hierarchy, which like the agent hierarchy allows descendant nodes to inherit grants. The Qualifier struct also has a fourth member, a string called "qualifier type", which indicates the type of struct that the Qualifier references. This member was included only to provide further information on qualifiers and has no effect on how grants are constructed.

A "function" ties the agent to a qualifier. It specifies the domain of authorization that the grant applies to, and is a member of an enumeration of functions that the Service Broker knows about. Functions need not apply to a particular qualifier object type. That is, the function "read experiment" need not have an Experiment object for a qualifier. This point will be clarified in the next section. Currently, the function enumeration includes a special "Super User" function. A grant that has this function does not need any qualifier associated with it, because it applies to *all* qualifiers. An agent that has this grant has all privileges on the Service Broker.

The Grant and Qualifier structs, and the function enumeration are detailed in [25].

4.2. Agent and Qualifier Hierarchies

Having discussed the Grant struct, which is the basic unit of authorization, we move on to describe how authorization policy in the iLab Service Broker is determined.

A user or subgroup that is a member of another group is referred to as a *descendant* of that group. Equivalently, the group is referred to as an *ancestor* of a user or a subgroup that belongs to it. Group memberships form an implicit agent hierarchy, such as the one shown in Figure 6. It is possible for a User to be a member of more than one group, for example, the User with ID "Mike" is a member of the "Course 6.012" and "Course 1.00" groups. Users that have memberships in more than one group, will have all their parent groups listed in the "Effective Group" page of the iLab web application (refer to Figure 4). Those users will then select an effective group, and assume the role that comes with that group for the entire web session. The user "Mike", for instance, will either choose a "6.012 Students" role or a "Course 1.00" role, and will proceed to the

"My Clients" page that will list all the clients associated with the role he has chosen.

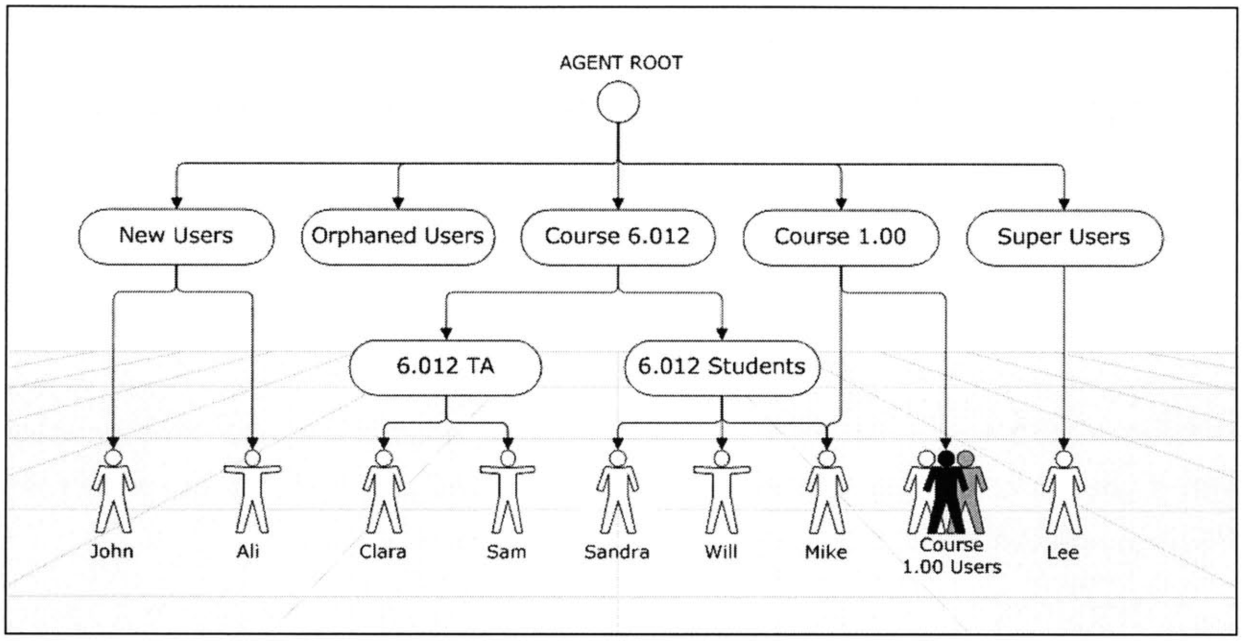


Figure 6. A representative agent hierarchy showing groups, subgroups and users.

Qualifiers also form a hierarchy, but unlike the agent hierarchy it is explicitly defined. The Qualifier struct has an array of qualifier parents as one of its members and these are assigned by administrators using methods of the Authorization API. The qualifier hierarchy is unlike its agent counterpart in another sense; its nodes reference not just User and Group objects, but also other types in the iLab architecture, such as Lab Server, Lab Client and Experiment objects. Figure 7 is a rendering of a qualifier hierarchy. Here, we have Experiment qualifiers that are children of User qualifiers, and Lab Client qualifiers that are children of a Group qualifier. These parent-child relationships represent some sort of association between the two types, such as "Sandra's experiment reports" or "lab clients used to teach course 6.012". It's possible to have these different relationships because Qualifier objects are type-less.

These hierarchies are useful for the authorization model because they allow descendants to inherit grants that have been applied to their ancestors. Grants can be applied to nodes at any level of the hierarchy. If a grant is applied at some node higher up in the hierarchy, it will span a greater number of descendant nodes than one that is applied at some lower level. In this manner we can control how granular a particular authorization should be.

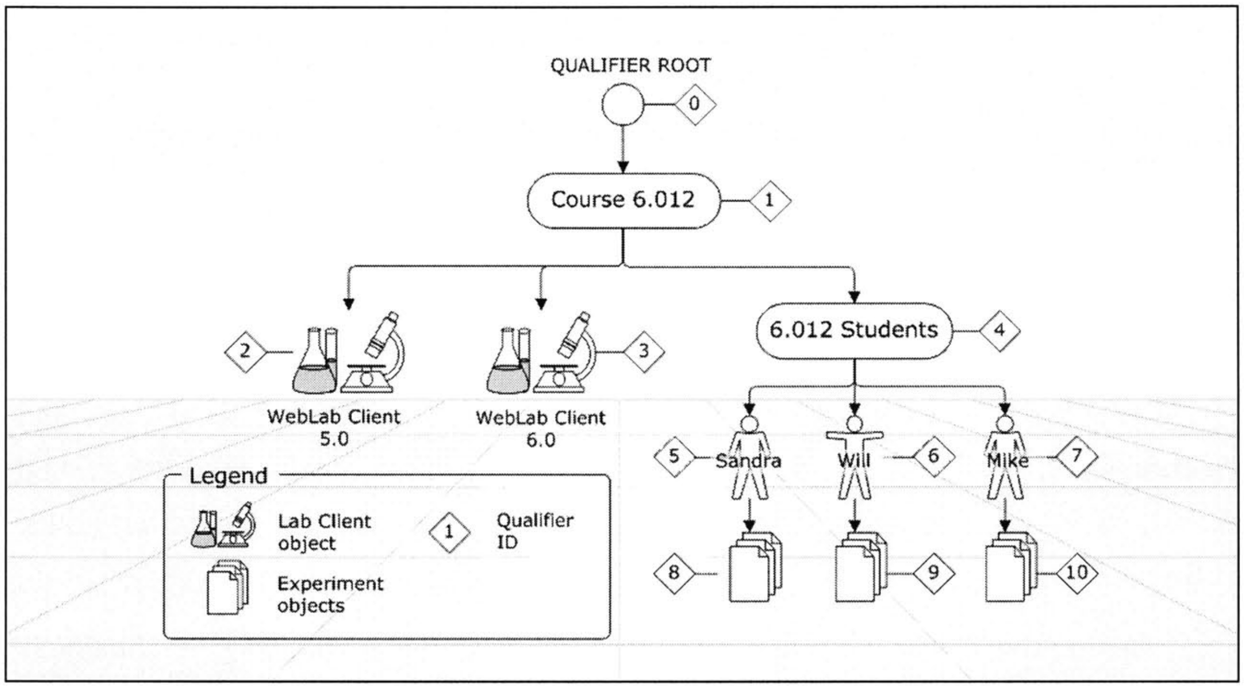


Figure 7. A representative qualifier hierarchy showing groups, users, experiments and lab clients.

Let's consider how grants can be applied to our example hierarchies. Assume that the grants in Table 1 have been declared. The first grant authorizes the "Course 6.012" group to use the lab client having a qualifier ID "2". By referring to Figure 7, we determine that this is the "WebLab Client 5.0" qualifier. This first grant is an example of a "coarse" grant, for it gives five different users (that is, the members of the "6.012 TA" and "6.012 Students" groups) the permission to use a lab client. The next grant gives the teaching assistants of course 6.012 permission to read the experiments of the students of course 6.012. The last grant gives Sandra the privilege to use the "WebLab Client 6.0" client, say because she is a graduate student. This is an example of a "fine" grant that gives only one person a specific privilege.

Table 1. Examples of Grant Structs.

Grant ID	Agent ID	Function	Qualifier ID
15	Course 6.012	Use Lab Client	2
16	6.012 TA	Read Experiments	4
17	Sandra	Use Lab Client	3

The grants in Table 1 are referred to as "explicit" grants because they have been declared explicitly. Inherited grants, such as Will's privilege to use the WebLab 5.0 client by virtue of the first grant, are referred to as "implicit" grants.

4.3. Authorization Policy

The Authorization API has a dual purpose; it defines the structs and mechanism that govern how authorization policy is enforced in the iLab architecture; it also presents a number of methods that allow authorization policy to be configured.

When the Service Broker infrastructure is initially installed, its business logic makes permission checks in accordance with a default authorization table, which is given in [25]. This table does not list actual grant structs, but rather pairs each business logic method with a "function designation". These designations include the members of the function enumeration that typically make up a grant struct, in addition to a number of other designations as described below. Table 2 presents a sample of these method-function designation pairs.

In its initial state, the Service Broker has only two effective user groups, "lab user" and "super user". Users that log in as super users are presented with the administrative pages shown in Figure 4, and can perform the most privileged functions. For example, Table 2 shows that to add a lab server, a user must have the "super user" designation. The super user designation implies the union of all other function designations. Other administrator groups will have a more restricted set of administrative functions.

Methods that have a "trusted" designation, such as the "authenticate" method, need not be preceded by an authorization check because these methods are not called directly from a web service and will actually be executing in trusted code. On a similar note, the "anyone" designation authorizes any user to call the associated method, and hence no authorization check is needed. The "owner" designation simply implies that the relevant method can only be invoked on the caller's own resources.

Table 2. Sample of Method-Function Designations.

Method API	Method Name	Function Designation
Administrative	Add lab server	Super User
Authentication	Authenticate	Trusted
Authorization	Find Grants	Anyone
Administrative	Save Client Item	Owner

4.4. Authorization Methods and Searches

Agent and qualifier hierarchies are modeled in the database using tables with rows that correspond to parent-child pairs. That is, each entry in these tables forms an edge of the agent or qualifier graph. This data model is discussed in further detail in Chapter 5. In practice, these hierarchies tend to be shallow, consisting of only a few levels, but are often broad, with each level comprised of many different nodes. This is especially apparent in groups that consist of many student users.

The Authorization API contains a number of methods that traverse these hierarchies to determine whether a particular grant exists or not. Searches are performed for both implicit and explicit grants, but because these hierarchies are usually broad, the majority of grants tend to be implicit. Consider the following use-case pertaining to the grants of Table 1: We are trying to determine whether the user Clara, who is a course 6.012 TA, has the permission to read Will's experiment report (refer to Figure 6 and Figure 7) That is, we need to determine the existence of a grant, implicit or explicit, of the form

{Clara, Read Experiment, 9}

where "Clara" is the agent ID and "9" is the qualifier ID of Will's experiment report. A search performed on the explicit grants of Table 1 yields no results. Now to establish whether an implicit grant exists, we need to check whether an explicit grant has been specified from which the grant in question can be implicitly derived. We first create a list of the agent's ancestors, and another list of the qualifier's ancestors. Then, for every member in each list we determine whether an explicit grant of the form

{agent ancestor, Read Experiment, qualifier ancestor}

exists. We do, in fact, find an explicit grant that satisfies this condition, and this is the second grant in Table 1. The group "6.012 TA" is an agent ancestor of the user "Clara", and the group "6.012 Students", with a qualifier ID of 4, is a qualifier ancestor of Will's experiment object.

The methods of the Authorization API that were implemented in the initial prototype allow us to manipulate Grant and Qualifier types, and make authorization checks as follows:

1. Grants: Add a new explicit grant by specifying an agent, function and qualifier tuple; Remove an existing explicit grant; List all existing explicit grants, or find specific ones.
2. Check authorization by specifying an agent, function and grant tuple; both explicit and implicit grants are checked.
3. Qualifiers: Add a new qualifier by specifying a reference type and parent qualifiers; Convenience methods to construct and manipulate the qualifier hierarchy.

The specification given in [25] provides a comprehensive listing of structs and methods in this API.

Chapter 5. The Service Broker Data Models

This chapter presents and discusses the structure of the backend database that supports the initial iLab Service Broker prototype. The database serves a number of purposes in the iLab architecture. Specifically, it stores: authentication and authorization data against which user actions are validated; user data to allow for persistence across multiple web sessions; and experimental results.

The data models given in this chapter have been conditioned by iLab project's initial focus on the batched experiment mode, and may not meet the requirements of interactive experiments, which are discussed in Chapter 6.

The data models are presented in terms of a SQL relational model, and consist of about 20 tables. These tables, or entities, have been divided into three broad categories: administrative, authorization and experiment storage. Each category of entities forms a standalone data model in which explicit relationships between the tables have been defined. Relationships between entities in different categories do exist, but they are few and are not bound by any referential integrity rules. One such example, previously mentioned in section 3.5, is the "experiment information" object that contains some administrative information pertaining to the person who conducted the experiment. If that person is deleted from the system, his / her associated experiment records will not be purged.

To reiterate from section 3.4, the backend database is accessed by a single fixed and trusted identity, that of the Service Broker. Authorization rules that govern what end-users can access are enforced at a higher-level tier, as illustrated in Figure 5, and the Service Broker executes stored procedures on the database accordingly.

5.1. The Administrative Data Model

The administrative data model encapsulates all information relevant to User, Group, Lab Server and Lab Client objects, which are acted upon by methods in the Administrative and Authentication APIs. The administrative data model has been broken down into smaller groups of entities in the following discussion for clarity.

5.1.1. Authentication Tables

Methods invocations in the Authentication API will act on or retrieve the information stored in the three tables shown in Figure 8.

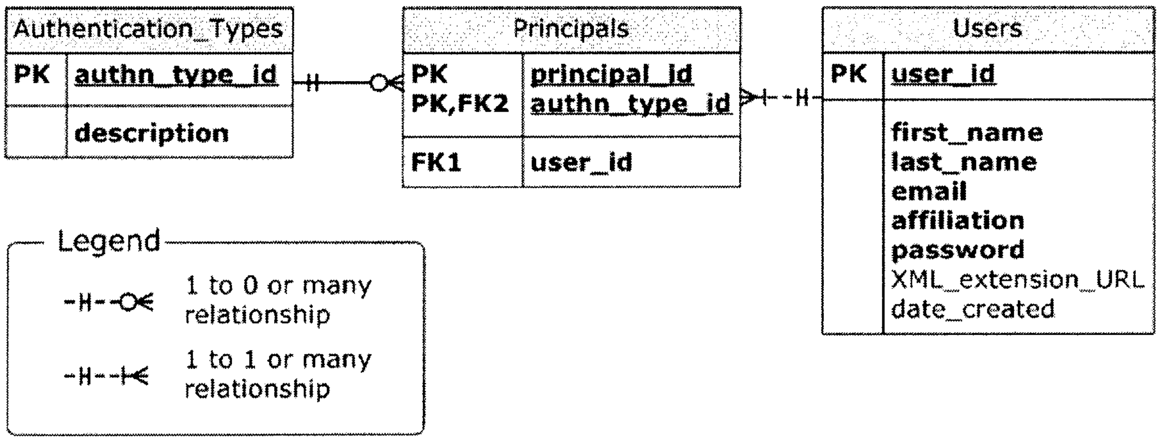


Figure 8. Authentication tables in the administrative data model. Fields in bold must have non-null values; PK = primary key, FK = foreign key

Let's first describe the "Users" table. This table stores a minimal amount of user-related information to avoid having 'fat' tables that contain many fields, which some lab owners may find redundant and store null values in them. If lab owners wish to include additional information about their lab users, they may use the extensible "XML_extention_URL" field. Going back to a point that was made in section 3.2, user-password associations are maintained only in the "Users" table of the database. That is, User objects in the higher API levels have no knowledge of passwords to avoid the possibility of passwords being intercepted during method invocations.

Although the current Service Broker implementation offers only a native authentication mechanism, it's possible to have other external mechanisms and each will have an entry in the "Authentication Types" table. Users can have multiple principal IDs but each identity must correspond to a different authentication mechanism.

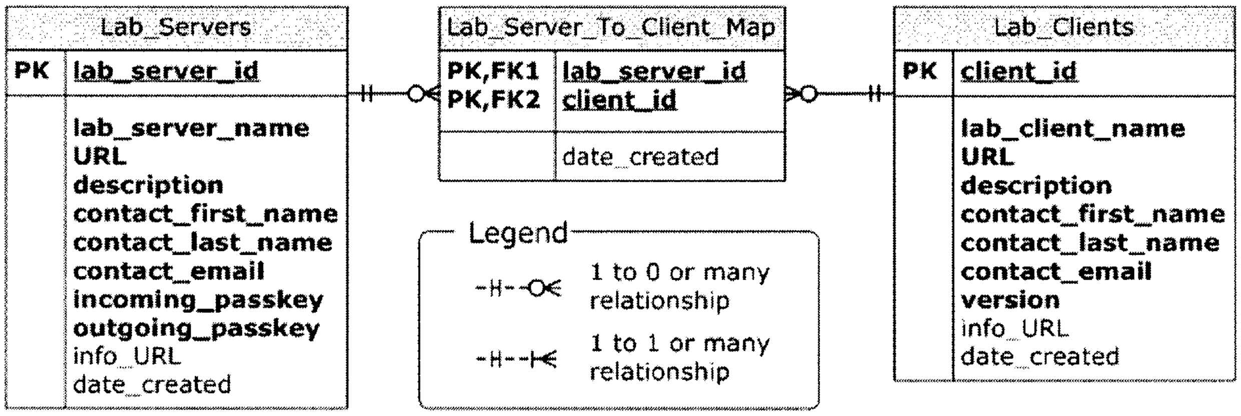


Figure 9. Lab server and lab client tables in the administrative data model. Fields in bold must have non-null values; PK = primary key, FK = foreign key

5.1.2. Lab Server and Lab Client Tables

Lab server and client tables are shown in Figure 9. Multiple lab clients can communicate with a lab server. These could be clients at various stages of development (hence the "version" attribute) or they could be very distinct clients that interact with a lab server in different ways. Though an uncommon scenario, the architecture also allows multiple lab servers to talk to a single lab client. This might be the case, say if a client communicates with a "simulation" server, in addition to the actual lab server. These client-server associations are modeled using a "Lab-Server-to-Client" mapping table that effectively splits the many-to-many relationship into two one-to-many relationships.

The "URL" field in the "Lab Servers" table refers to the URL that points to the Service Broker-to-lab server web service that resides on the lab server. The "URL" field in the "Lab Clients" table refers to the URL that points to the web page that hosts the lab client. The "info_URL" field is a placeholder for a web page that provides useful information about the iLab component, such as frequently asked questions or a user manual.

5.1.3. Data Storage Tables

Users are allowed to store two broad categories of data on the Service Broker, client items and experiments. The associated tables are shown in Figure 10. Client items can refer to a number of things, from a saved experiment specification, to a lab client preference that the user has indicated, such as how plots should be displayed. These items are stored as name-value pairs in the "Client Items" table. Experiments are a 'special' sort of client item, which are stored in a different table, and are discussed as part of the experiment storage data model in section 5.3. The "Stored Item Summary" table is a collection of attributes relevant to the stored data item, such as the associated user, group, and lab client. The Service Broker readily understands this information, whereas client items and experiments are opaque to it.

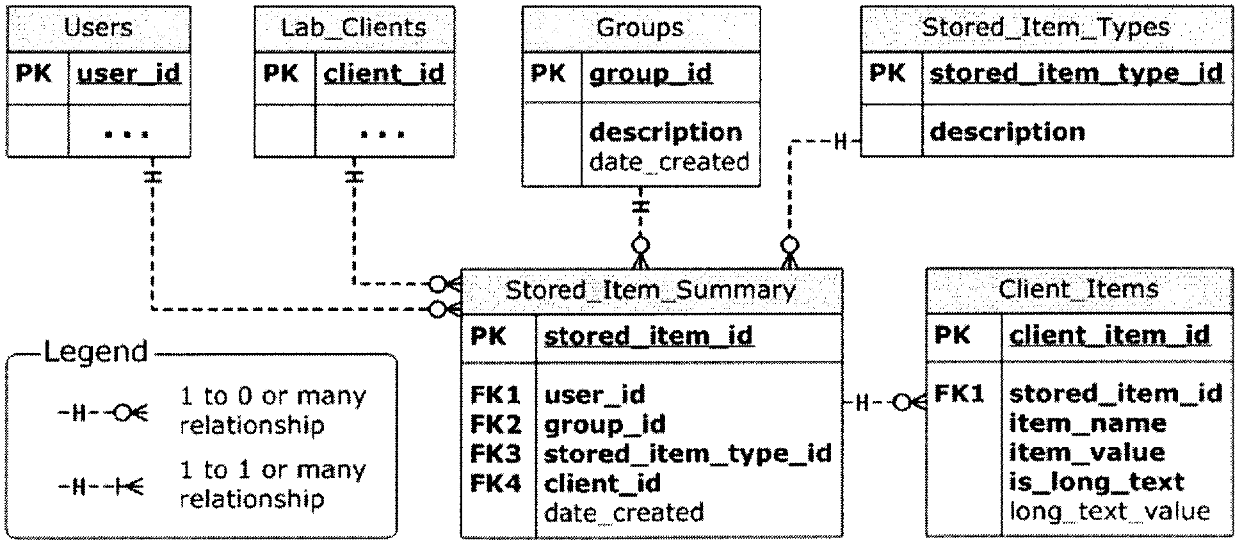


Figure 10. Data storage tables in the administrative data model. Fields in bold must have non-null values. Attributes of the "Users" and "Lab Clients" tables have been omitted; PK = primary key, FK = foreign key

5.2. The Authorization Data Model

The authorization data model is shown in Figure 11. At the center of the model is the "Grants" entity that stores all explicit grants. This table references a single agent, qualifier and function. The "Functions" table is a validation table whose entries correspond to the enumeration of functions discussed in the previous chapter.

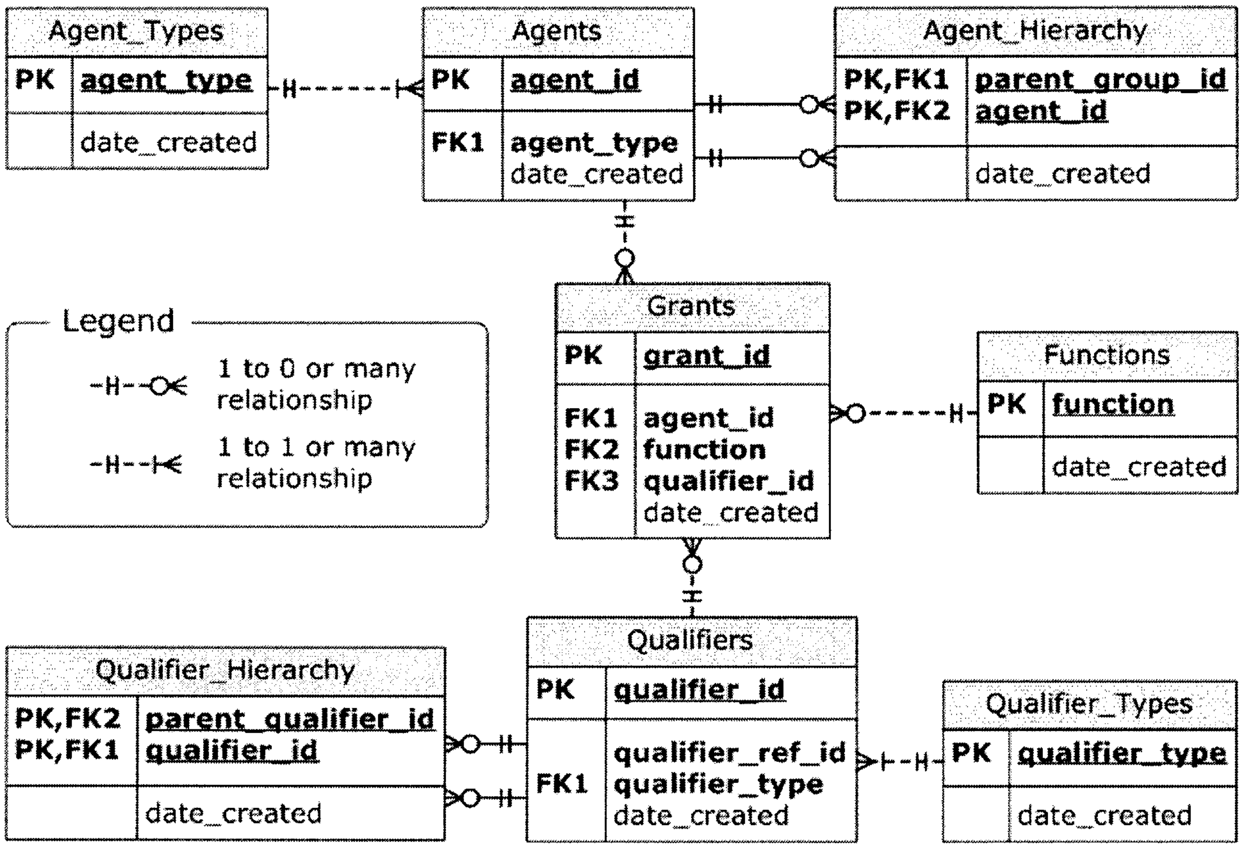


Figure 11. The authorization data model. Fields in bold must have non-null values; PK = primary key, FK = foreign key

Agents are references to User and Group objects, and agent ID is a placeholder for either a user or group ID. Since each agent ID must be unique, this implies that each member of the set of all user and group IDs must also be unique. We enforce this uniqueness constraint by inserting newly created user and group types in the "Agents" table before inputting them into their corresponding tables. Because there is no referential integrity between the "Agents" table and the "User" or "Group" tables, we must ensure that the creation (or removal) of a user or group record in (from) two separate tables is handled programmatically.

The "Agent Hierarchy" table provides information on all group memberships in the system. These memberships are listed in the table in the form of 'user-group' and 'subgroup-group' pairs. This table is used to bestow implicit authorizations on 'descendant agents' as described in Chapter 4. The child-parent ID pairs in this table each form a two-part key.

Fields of Qualifier structs are stored in the "Qualifiers" table. The "qualifier_ref_id" is the ID of the resource that the qualifier points to. Valid qualifier types are lab clients, lab servers, users, groups and experiments. To repeat a point made in the previous chapter, the "qualifier_type" field is purely informative – it does not affect how grants are constructed. Like the "Agent Hierarchy" table, the "Qualifier Hierarchy" table records consist of child-parent qualifier pairs that form a two-part key. This hierarchy serves the purpose of allowing agents to inherit privileges to access certain 'descendant resources'.

5.3. The Experiment Storage Data Model

Section 5.1.3 mentioned that users could save two types of data on the Service Broker, client items and experiments. This section describes the tables pertaining to the storage of experiment records, and these are given in Figure 12. Experiment records typically consist of three parts that are opaque to the Service Broker: the lab server configuration at the time the experiment was conducted, the experiment specification submitted to the lab server, and experiment results returned by the lab server. We can see from the "Experiments" table that the field relevant to lab server configuration must be non-null. However, experiments may or may not require a specification or return results, and this is indicated by the binary "uses_expt_specification" and "uses_expt_results" fields. Administrative data like "user ID", "group ID", and "lab client ID", associated with a particular experiment, is retained in the "Stored Item Summary" table, which was described as part of the administrative data model section. The "stored_item_id" field of the "Experiments" table (this has been circled in Figure 12) maps onto the "stored_item_id" field of the "Stored Item Summary" table. Again, no referential integrity has been enforced between these two tables because they span different data models.

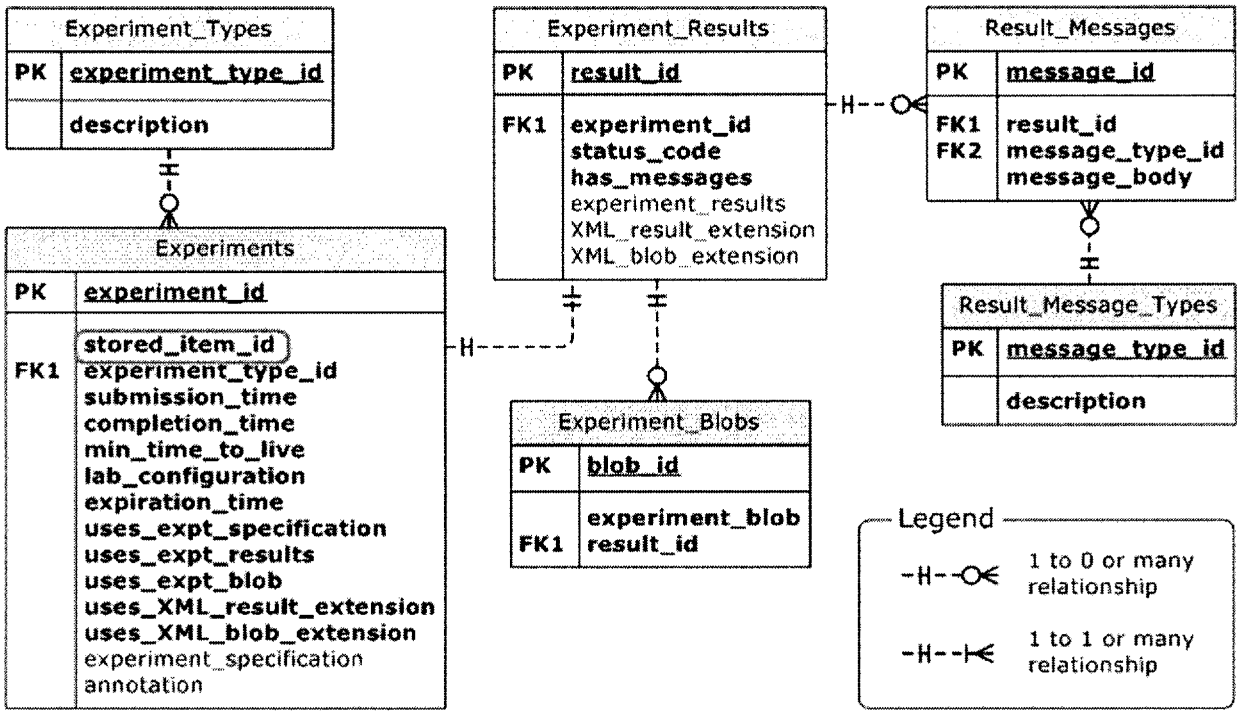


Figure 12. The experiment storage data model. Fields in bold must have non-null values; PK = primary key, FK = foreign key

An experiment record can have a single "Experiment Result" record associated with it. The fields of the "Experiment Results" entity were discussed in section 3.5, the Experiment Storage API. An experiment results record can have zero or more "Result Messages" or "Experiment BLOBs". Result messages can be either validation warnings and errors, or execution warnings and errors. Error messages are issued by experiment specifications that cannot be accepted by the lab server, whereas warning messages are issued by specifications that satisfy some criteria set by the lab server owner.

5.4. Accessing Data in the Database

The internal database API is a layer that sits between the business logic APIs and the DBMS, as shown in Figure 5. The methods of this API are called only by trusted code. These methods declare parameters that correspond to the members of the various iLab objects. They then establish an authenticated connection with the DBMS and ship off these parameters to stored procedures that act to alter the state of, or retrieve information from, the database.

We chose to use stored procedures to execute SQL commands on the database rather than embedding SQL statements in our code for a number of reasons: to take advantage of the performance gains associated with stored procedures since they are compiled only the first time they are executed, and this compiled version is stored in memory to process subsequent calls; to minimize the amount of traffic on the network since less text is sent by the application to the stored procedures on the database server; and to be able to access data from the database securely. Additionally, stored procedures provide a layer of indirection between our application and the database design, which is useful if the database design needs to be changed. In this case, the application layer can remain totally unaware of the change as long as the stored procedures are modified to return the result set that the application expects [26].

Certain stored procedures make use of transaction processing to execute a batch of SQL commands. In a transaction, changes to the database are not committed unless all the SQL commands in a batch execute successfully. If any one of the commands issues an error, then the transaction is rolled back and all changes are undone.

The iLab's authorization mechanism makes frequent checks against the information contained in the authorization tables. Since database connections are expensive we use cached "DataSet" representations of these tables on the Service Broker to update and retrieve authorization information. The DataSet class is exposed by the ADO.NET architecture, which provides an object-oriented view into the database. These DataSet objects are cached on the application server without requiring a continuous connection to the database server. These objects will only make a connection when a change is made to them, and that change needs to be reflected in the database, or if a change is made to the database and the DataSets need to be updated [10].

Chapter 6. The iLab Interactive Experiment

This thesis has focused thus far on the batched experiment mode. Our survey of Internet-accessible labs, however, has shown that they predominantly involve interactive experiments. A batched experiment typically requires the user to hand off an experiment specification to a lab server in a "load-and-go" fashion, and refer back once in a while to check whether the results are ready. An interactive experiment, on the other hand, requires the user to monitor, and perhaps respond to changes in the state of the lab server. The experiment often entails other user activity, such as deciding which results need to be recorded and performing post-experiment clean up. Interactive experiments are probably more representative of physical laboratory experiments.

The initial iLab prototype provided much insight into what was required to help lab owners bring their experiments online. That prototype, however, presents a number of impediments that might deter *interactive* lab owners from integrating their systems within the iLab architecture. What is likely to be of most concern to them is the prototype's topology in which the Service Broker acts like a tight coupling between lab server and client. This arrangement forces messages that are destined for either the lab server or client to traverse the Service Broker, and become marshaled / unmarshaled into XML along the way. This extra hop introduces additional latency between the client and lab server.

Interactive lab owners want a mechanism that allows our Service Broker to establish a secure connection between the lab client and server, and then to move out of the way, while the two components communicate independently. This chapter discusses the Remote Polymer Crystallization interactive lab and how the iLab infrastructure was modified to allow the lab client and server components to communicate directly and securely via some domain-specific protocol.

6.1. The Remote Polymer Crystallization Lab

The online remote polymer crystallization experiment involves the heating of a polymer sample past its melting point, then allowing it to cool under controlled temperature conditions. This results in the formation of a number of crystal nuclei, which grow in diameter with the passage of time. By observing these nuclei through a polarized light microscope, and analyzing them, students can learn about the crystallization kinetics of the polymer sample.

The student interacts with a Java applet lab client that communicates directly with the remote microscope lab server over TCP/IP sockets. The applet provides a number of controls that allow the student to view images of the sample and to manipulate various hardware settings, such as the temperature and position of the stage upon which the sample is heated. Additionally, the user can tell the applet when to start recording images of the sample and when to stop. Experiments, therefore, can consist of multiple runs in a single session, with each run generating its own result set [27].

The remote microscope lab server, which interfaces to the various hardware components, is written in Python. The hardware consists of an imaging microscope, a digital camera, and a movable heating stage. The lab server accepts messages from the applet on an open socket connection, and converts them into a form that can be understood by, and used to control these components. The server also interfaces to a database where experiment information is stored and to a file system where images from the microscope are saved [27].

The Java applet and Python server together can be thought of as "the remote microscope". As Talavera has pointed out in his Master's thesis, the remote microscope is a two-tier architecture, with one user in control of the server in any single session [27].

Multiple users can, however, interact simultaneously with this online lab via another component, the "framework server". The framework server allows users to manage their accounts, reserve a time slot on the remote microscope and review and analyze past experiments. It is a three-tier architecture, with a web-based interface, business logic, and database. This component writes to, and reads from the same database that is accessed by the microscope server. This allows the microscope server to verify certain parameters, like those pertaining to a reservation that have been set by the framework server and passed to it via the client [27].

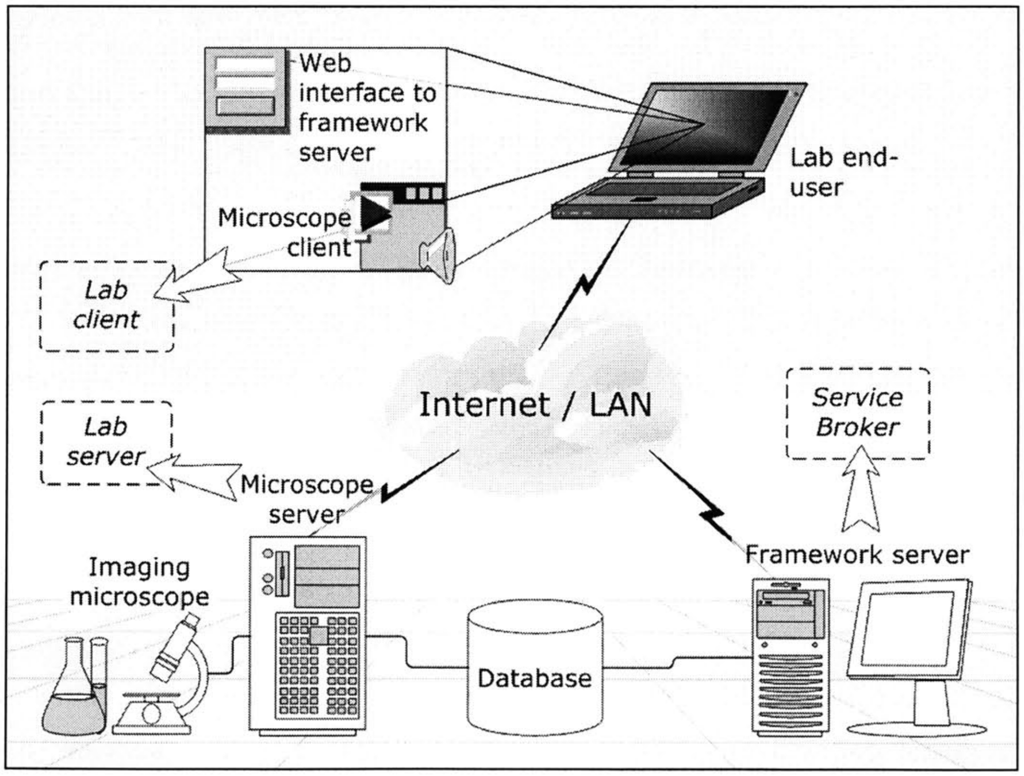


Figure 13. The remote polymer crystallization lab topology. Corresponding iLab components are given in dashed boxes.

It is clear that if the remote polymer crystallization lab were to adopt the iLab architecture, then the Service Broker would assume the role of the framework server, and perform many of its functions. Refer to Figure 13. Aside from an authenticated client-to-lab server connection, this Internet-accessible lab would also require a stand-alone experiment storage service that the lab server could write results to directly.

6.2. General Ticketing and Process Agents

One aspect of the remote polymer crystallization lab that was of particular interest to us was how the lab client was able to authenticate itself to the lab server, and convince it that it has a reservation to run an experiment. The framework server exposes a user interface that allows users to make a reservation on the lab server; this reservation information is stored in the database. Once it's time for a client to run an experiment, the framework server passes it a "reservation token" that also includes the user's GUID. This token is then presented to the lab server, which checks the information contained in it against the database. If the information is valid, the lab server opens up a socket connection for the client, and the experiment can proceed.

This mechanism motivated us to devise a "general ticketing" scheme on the Service Broker that would allow it to write out tickets and dispense them to clients. These tickets would be used by clients to access the different resources (or entities) advertised by the Service Broker, with lab servers being one type of resource. Various types of information can be included in a ticket, including authentication, authorization, and scheduling information. The entity which redeems the ticket will use it to learn this information about its client, the ticket holder. The concepts underlying general ticketing are discussed in section 6.4. In addition to tickets, there are two other requirements to allow the lab client to establish a direct connection with the lab server. These are (1) a service to which the lab server can write experimental results and from which clients can retrieve, and (2) a service to schedule time on the lab server. These two entities are the

"Experiment Storage Service" and the "Scheduling Service" respectively, the subjects of section 6.3.

The ideas that were raised in the preceding paragraph illustrate some of the requirements of interactive experiments. These requirements force us to depart from the initial iLab architecture paradigm, where all services required by an online lab are hosted on the Service Broker, to one where the services are broken out as stand-alone entities. Our ticketing scheme is "general" in that tickets are issued not only to clients who want to use a lab server, but also to clients who want to use an Experiment Storage Service or any other service advertised by the Service Broker. These stand-alone entities, with each entity providing a specific service, may be regarded as the Service Broker's *process agents* because they run on different systems.

One difference between the remote polymer crystallography lab's topology and that of the iLab architecture is that in the former case, the component responsible for writing reservation tokens (the framework server) shares a common database with the lab Server. This makes it easy for the lab server to determine the authenticity of any tokens it sees. In the case of the iLab architecture, the Service Broker stores the tickets it writes in its own "ticket store" database which the lab server has no direct connection to, since it is a separate process agent. The way a lab server will go about verifying the validity of tickets in the iLab architecture is by calling the Service Broker who wrote it via web services. This idea will be further expanded in section 6.4.

6.3. iLab Stand-Alone Services

The concept of "process agents" running on distinct systems enables clients to bypass the Service Broker on nearly all method calls. With a valid Service Broker-issued ticket, clients and other entities can establish a direct connection with a process agent offering a particular service. This section discusses two such agents, the "Experiment Storage Service" and the "Scheduling Service".

6.3.1. The Experiment Storage Service

The experiment storage service (ESS) provides much of the functionality that was catered for by the Experiment Storage API (section 3.5). However, that API treated experiment objects as atomic units, a notion that cannot be applied to experiments like the remote polymer crystallography lab, where a single client session may consist of multiple experiment runs. For this reason, an ESS experiment object is defined as an ordered set (or sequence) of experiment records and each record consists of a single payload. Payload types may vary from one record to another, that is, a payload can hold control information, error messages or string results. Once an experiment is "closed" by the user or process agent, or if an experiment timeout occurs, then records can no longer be added to it. Individual experiment records cannot be deleted or modified, since that would amount to altering the state of a previous experiment. Only whole experiments can be deleted [28].

The ESS provides a generic mechanism for storing large, binary experimental results, such as images. Rather than transferring the binary data to the ESS as an attachment in a synchronous web service call, the data is streamed asynchronously. However, prior to streaming the actual binary data, the producer of the data (which is another process agent) will create a BLOB record on the ESS that holds a byte count of, and a checksum for the binary data. The producer will use this information to determine the state of the data that has been downloaded on the ESS. Once the producer is convinced that the download is successful, it may choose to associate that binary data with an experiment record, in which case the BLOB becomes a permanent part of the experiment. For further information on the ESS, the reader is referred to [28].

6.3.2. The Scheduling Service

Labs such as the remote polymer crystallography lab require users to make prior reservations on the lab server. This was not the case with the WebLab batched experiment where experiment specifications were submitted to a queue and processed in a first-in first-out (FIFO) manner. For most labs, however, there exists an obvious need for scheduling to allocate time fairly among end-users.

The concept of "scheduling" is not consistent across all labs, and each lab owners tends to view the problem of scheduling differently. For example, lab owners may want to distribute their lab server time using a first come, first served basis, or they may want to use another allocation system, such as auctioning. Lab owners will also want to attach rules on how reservations are made, such as an upper limit on the length of time for a reservation. To further add another dimension to the problem of scheduling, lab owners are not the only party involved in setting scheduling policies on a lab server. For instance, Service Broker administrators may also want to attach rules on how users can sign-up for time that is allocated by a particular lab server. For additional information on the mechanics of the Scheduling Service, the reader is referred to [29].

6.4. General Ticketing Concepts

This section describes the general ticket object definition, and the two broad categories of tickets, "temporary tickets" and "permanent tickets". It also gives a discussion of the data model and web services used in a prototype that demonstrates general ticketing.

6.4.1. The Ticket Object

"Ticket" objects are a representation of tickets issued by the Service Broker. Every issued ticket has a unique ticket ID, and a randomly generated passkey, as well as some other fields. The ticket ID and passkey constitute the "ticket stub". A number of Ticket object fields hold the IDs of the various parties involved with the ticket. These are the entity that wrote the ticket (the "issuer"), the entity where the ticket is to be redeemed (the "redeemer"), and the entity that requested the ticket to be written (the "sponsor"). The issuer is *always* a Service Broker entity, whereas the redeemer and sponsor are usually process agents. These ticket instances are immutable, and the ticket "master" copy resides on the issuer, or Service Broker entity. Only ticket stubs are exchanged between clients and process agents.

When a client submits a ticket stub to a redeemer process agent, the redeemer will ask the Service Broker to verify whether the ticket stub is valid. This is accomplished through a web service call. If the stub is valid, the issuer will retrieve the associated ticket and hand it over to the process agent. For example, suppose a lab client shows up at a lab server with the stub of ticket X. The lab server will then call a "verify ticket" method on a web service exposed by the issuer of the ticket before granting its client a direct connection. The ticket ID and passkey are passed as arguments to the web method in a SOAP-request message. The issuer will look in its ticket store, and if it finds a matching ticket, instantiate a ticket object, set its fields to those of ticket X and ship it off to the lab server in a SOAP-response envelope. In this manner, the lab server will have an updated copy of ticket X that contains all relevant information about its client. If the issuer does not find a matching ticket in its ticket store, it will return a "null" value, and the lab server will not to grant its client a direct connection.

Tickets that have been issued by a Service Broker may be modified or cancelled at some point in the future. However, since ticket instances are immutable, the issuing Service Broker needs to (1) destroy the old ticket, and (2) inform the redeemer process agent of the modification or cancellation if the ticket was already redeemed. In the event of a ticket modification, the issuing Service Broker will additionally create a new ticket, with updated information, and put it in its ticket store.

Ticket objects also contain an XML field called "payload", which holds information that is understood by the redeemer process agent. For example, the payload field may contain "start-time" and "end-time" tags which a lab server interprets, respectively, as the time it should grant its client a connection, and the time to terminate it. A payload field may be specific to a particular process agent type, and might not be understood by another. As such, a lab server ticket payload may be quite different from an experiment storage service ticket payload. Service Brokers might need to understand part of a payload because this field may state certain conditions that need to be met before the associated ticket is given to its client. For example, a Service Broker that issues a ticket with a payload that contains a "start-time" tag might need to wait until that time before it passes the ticket to its client. We opted to include an XML payload field in our ticket object over having a base-class ticket type from which specific tickets are derived. This is so that process agents can take advantage of the flexibility to extend their payload definition, as long as the issuer knows how to write it.

Payloads may contain tags that correspond to temporal fields. In order to avoid discrepancies due to localization, any temporal field is made in reference to the time on the redeemer process agent. Going back to our previous example, the client should be aware that their connection's "start-time" and "end-time" correspond to the time in the lab server's locality.

We need to distinguish between two types of tickets, "temporary tickets" and "permanent tickets". The tickets described thus far were examples of "temporary tickets" that are redeemed only once by a client at a redeemer process agent. After a temporary ticket has been redeemed, it can no longer be used again. In contrast, a permanent ticket, which is also referred to as a "Service Broker ticket", is a contract between process agents and the Service Broker. Permanent tickets serve two purposes. They are used by process agents to (1) authenticate themselves to the Service Broker and (2) to state who can sponsor their tickets. Recall that a ticket's sponsor is the entity that has requested the creation of the ticket. A ticket's issuer, on the other hand, is the entity that writes and stores the ticket, and is always a Service Broker entity. For example, a lab server may give a scheduling service entity the privilege to sponsor its tickets. Then, when clients show up at the scheduling service to request time-slots on the lab server, the scheduling service can ask the Service Broker to write and store tickets for them.

We developed a prototype that illustrates some of the concepts outlined above, and tests their use in the context of the remote polymer crystallography lab. Screenshots are given in Appendix B-I.

6.4.2. The General Ticketing Data Model

The data model pertaining to general ticketing is shown in Figure 14. This model resides on the Service Broker. The first table of interest here is the "iLab

Entities" table. This table lists all process agents, or entities that the Service Broker knows of. The types of entities listed in this table may be, but are not limited to, lab servers, experiment storage services, and scheduling services. Each one of these entities needs to host a web service or web application that clients can use to present tickets.

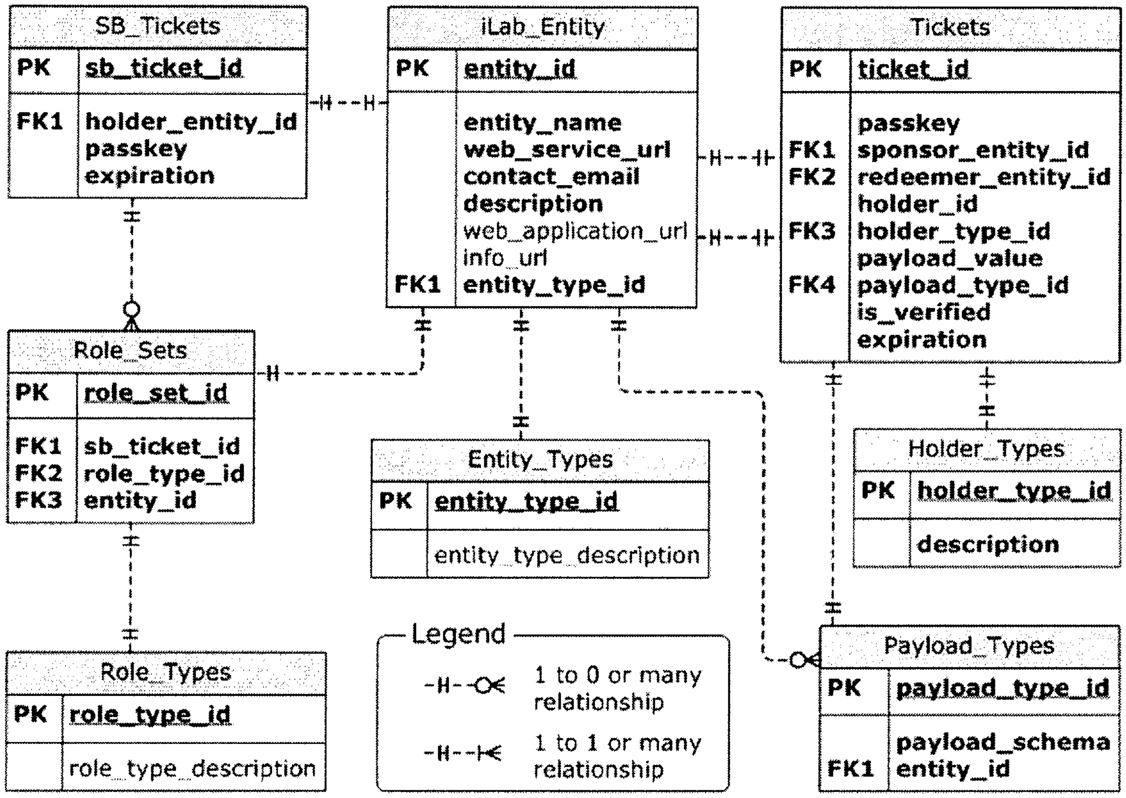


Figure 14. The general ticketing data model. Fields in bold must have non-null values.

The "Tickets" table holds all temporary tickets written by the Service Broker. Note that the "sponsor_entity_id" and "redeemer_entity_id" fields reference records in the "iLab Entities" table. Since the ticket issuer is always the Service Broker, that field was not included. The ticket "holder" can either be a user or a process agent entity, but the former case is the more common one. Although, we did not implement the following aspect in our prototype, tickets must typically reference a particular "payload type" as discussed earlier. The "is_verified" field is a binary field whose value is set to "true" if a process agent verifies a particular ticket. This allows the Service Broker to decide whether to inform a process agent that a ticket was cancelled or modified. The "expiration" field gives the date and time a particular ticket becomes invalid. This is useful for housekeeping tasks on the Service Broker, such as purging invalid tickets.

Permanent, or Service Broker tickets, are stored in the "SB Tickets" table. Each iLab entity has a single Service Broker ticket. This table does not include a "payload" field since the Service Broker needs to extract information from these tickets and it would not make sense to have to always decode that information from XML. Instead, this information is stored in the "Role Sets" table. Each role set consists of a "role type" and "entity ID" pair. The available role types are "Create", "Create and Give", and "Cancel". For example, suppose lab server Y wants to grant scheduling service X the privilege to create and cancel tickets on its behalf. It would ask the Service Broker to reference the role sets {"Create", Y} and {"Cancel", Y} to lab server X's permanent ticket.

6.4.3. The General Ticketing Web Service Methods

This section describes the web service methods that were implemented to put our general ticketing mechanism in effect. The first category of methods resides on a web service exposed by the Service Broker. They are:

1. Register: This method is called by a process agent that wants to become a member of the iLab architecture. It is typically called by entities such as lab servers, experiment storage services and scheduling services. By calling this method, the process agent is requesting a Service Broker ticket. If the Service Broker administrator is satisfied with the information provided by the calling entity, it will create a new entry in the "iLab Entities" table and write a new Service Broker ticket for that entity. Any entity is authorized to call this method.
2. Get registered entities: This method is invoked by a process agent that wants to learn about other entities registered with a Service Broker. The process agent must have a valid Service Broker ticket to call this method.

3. Grant / remove ticketing permission: A process agent will invoke these methods to manipulate the "role sets" associated with its Service Broker ticket. It will use the "Get registered entities" method to determine the various services available. The process agent must have a valid Service Broker ticket to call this method.
4. Create / cancel ticket: These methods are invoked by a process agent to create a new temporary ticket or invalidate an existing one. All relevant tickets parameters are passed as arguments to this method. The process agent is authorized to call this method only if it has a valid Service Broker ticket.
5. Verify ticket: This method is called by a process agent that wants to determine the validity of a temporary ticket ID / passkey associated with a particular client. The method returns a ticket object instance accordingly. As before, the process agent is authorized to call this method only if it has a valid Service Broker ticket.

The second category of methods resides on a web service exposed by the process agent. In our prototype the methods below were part of the lab server web service:

1. Install Service Broker ticket: A Service Broker will call this method in response to a registration request made by a process agent. It will pass the process agent its Service Broker ticket which will contain an ID and passkey. These two parameters represent that agent's authorization to call methods on Service Broker web service.
2. Get current time: This is an important method because it returns the current time on a process agent, which should be used in reference to any temporal field in the payload of a ticket to be redeemed on that agent.
3. Connect: Clients will call this method to request a direct connection to the process agent. In calling this method, a client will include a temporary ticket ID / passkey combination (ticket stub) in the header

of the method call. If the combination is valid this method will return a domain-specific string. If not, it returns a null value. In our prototype, this string is a password that is set by the lab server. The client will then transmit the password to the Python server on an open socket connection that is continuously listening to the client. A correct password will allow the client to transmit other commands on the socket connection, such as commands to control the lab hardware.

4. Cancel ticket: This method is called by a Service Broker to inform a process agent that a particular ticket has been modified or cancelled. Recall that tickets are immutable, and as such, any modifications to what a ticket entitles a client to do will result in the cancellation of the old ticket and the creation of a new one.

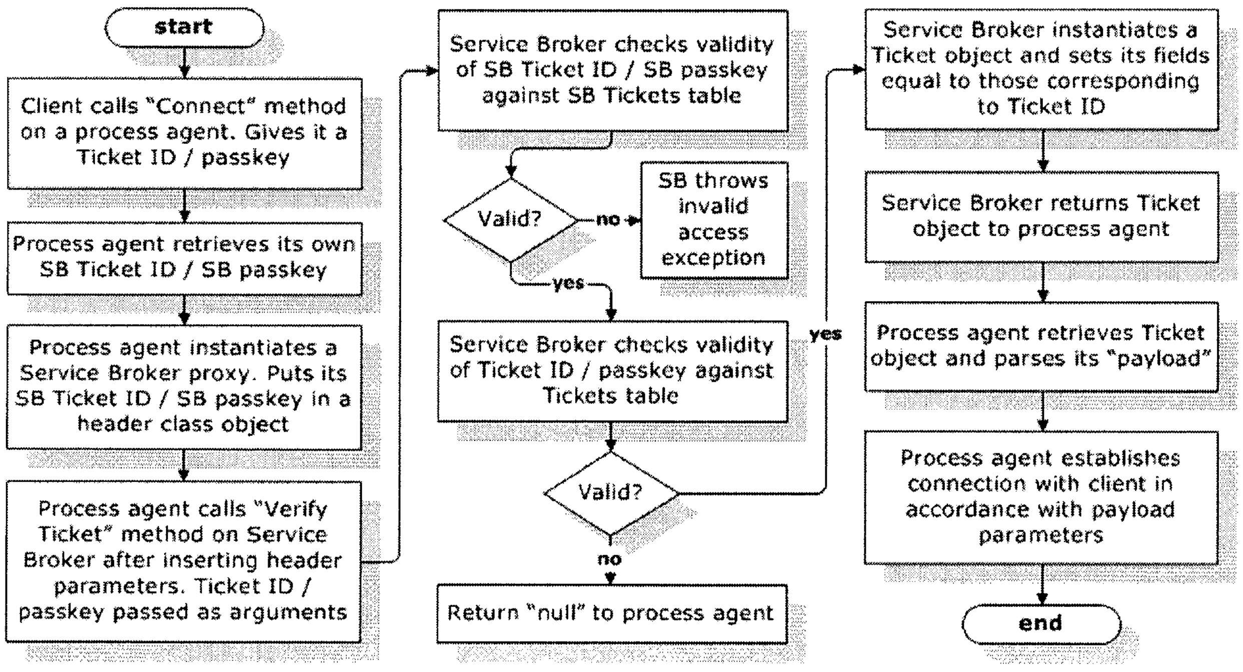

start
Client calls "Connect" method on a process agent. Gives it a Ticket ID / passkey
Process agent retrieves its own SB Ticket ID / SB passkey
Process agent instantiates a Service Broker proxy. Puts its SB Ticket ID / SB passkey in a header class object
Process agent calls "Verify Ticket" method on Service Broker after inserting header parameters. Ticket ID / passkey passed as arguments
Service Broker checks validity of SB Ticket ID / SB passkey against SB Tickets table
Valid?
no
SB throws invalid access exception
yes
Service Broker checks validity of Ticket ID / passkey against Tickets table
Valid?
no
Return "null" to process agent
yes
Service Broker instantiates a Ticket object and sets its fields equal to those corresponding to Ticket ID
Service Broker returns Ticket object to process agent
Process agent retrieves Ticket object and parses its "payload"
Process agent establishes connection with client in accordance with payload parameters
end

Figure 15. Process flow diagram showing processes and decisions involved in resolving a client request for a process agent connection.

With general ticketing, we have eliminated the need for a lab client-to-Service Broker API. This mechanism uses web services to set up a direct client-to-server communication link, and all subsequent communication will occur using some domain-specific remote procedural call (RPC) or messaging middleware.

Figure 15 is a process flow diagram that depicts the sequence of events that occur when a client requests a direct connection to a process agent.

Appendix B-II provides a listing of method signatures and object definitions pertaining to the general ticketing mechanism.

6.5. General Ticketing on the Experiment Storage Service

Our general ticketing mechanism may be used to administer access to the ESS. An ESS ticket payload will consist of one or more "roles", where each role is a level of access that the holder of the ticket has on the ESS. Examples of roles are "create experiment", "write experiment", and "read experiment". These roles may apply to specific experiments, or they may apply to all experiments in general. Since a Service Broker will issue the ESS tickets, it will need to understand how to write these roles. Again, the ESS may want to grant another process agent the privilege to create tickets on its behalf by manipulating that process agent's permanent ticket.

6.6. Unresolved General Ticketing Issues

Our interactive lab prototype answered only the most basic questions of general ticketing. However, there are a number of issues surrounding general ticketing that still need to be resolved. For instance, it's not clear as to what portions of the ticket payload the Service Broker is expected to understand, or how those portions are defined such that they can be applicable to particular process agent types. Another issue is whether the mechanics of the "connect" method implementation are applicable to all lab servers. These questions can be answered with the implementation of the experiment storage service, scheduling service, and other interactive labs.

Chapter 7. Future Directions and Limitations

The iLab project has undergone a number of development cycles since its inception, with each round tackling new facets of Internet-accessible labs. Starting off with the WebLab batched experiment, we explored what common services Internet-accessible lab users sought. We identified these as authentication, authorization, lab administration, experiment storage and scheduling, and grouped them together in a single entity, the Service Broker. Lab administrators and users consumed these services through a number of web interfaces exposed by the Service Broker. Certain web pages permitted administrators to configure policies concerning the operation of their labs, while other pages allowed lab users to manage their accounts and launch lab clients.

The initial prototype also shed some light on the mechanics of web services, and their relevance to this project. The WebLab client, a Java applet, invoked a .NET web service on the Service Broker in order to communicate with the lab server. This attested to and convinced us of the interoperability of web services, which was an important consideration for the iLab architecture since its components were unlikely to all be running identical operating systems.

The next phase of the project deals with interactive experiments. A careful study of the Remote Polymer Crystallography lab revealed that the iLab architecture, as designed for the batched experiment model, was not well-suited for interactive experiments. This was largely due to the fact that the common services were part of the Service Broker business logic, and as a result all client-to-server communication was routed through the Service Broker. By breaking out services like data storage and scheduling, and using tickets to transmit authentication and authorization information, we reduced the number of messages passed through Service Broker during the course of an experiment to a minimum.

The iLab Service Broker is steadily approaching a state where it can be deployed in multiple campuses. Currently, the architecture is being reviewed by a number of offshore universities, including Chalmers University of Technology in Sweden and the American University of Beirut in Lebanon. One issue that requires further investigation is how the underlying communications infrastructure supporting these institutions will affect the online laboratory experience for their end-users. Our prototypes assumed the availability of a broadband connection, but many potential end-users are still connected to the Internet via some lower bandwidth medium, such as phone lines.

The iLab architecture has yet to address sensor lab experiments. Online sensor labs are important because they allow for the remote observation of real-world engineering systems, such as physical infrastructure, that cannot easily be studied in a traditional lab setting [3]. Chapter 1 briefly talked about sensors as devices that produce continuous data streams. The execution of queries over these data streams, to extract meaningful information from them, is not an easy task. This is because streaming sensor data tends to be noise-ridden, unprocessed and sometimes delivered at unreliable rates. As such, sensor data sources are fundamentally different from highly-engineered sources that most DBMS systems are used to. Additionally, sensors are push-based devices, whereas regular DBMS systems perform pull querying [11].

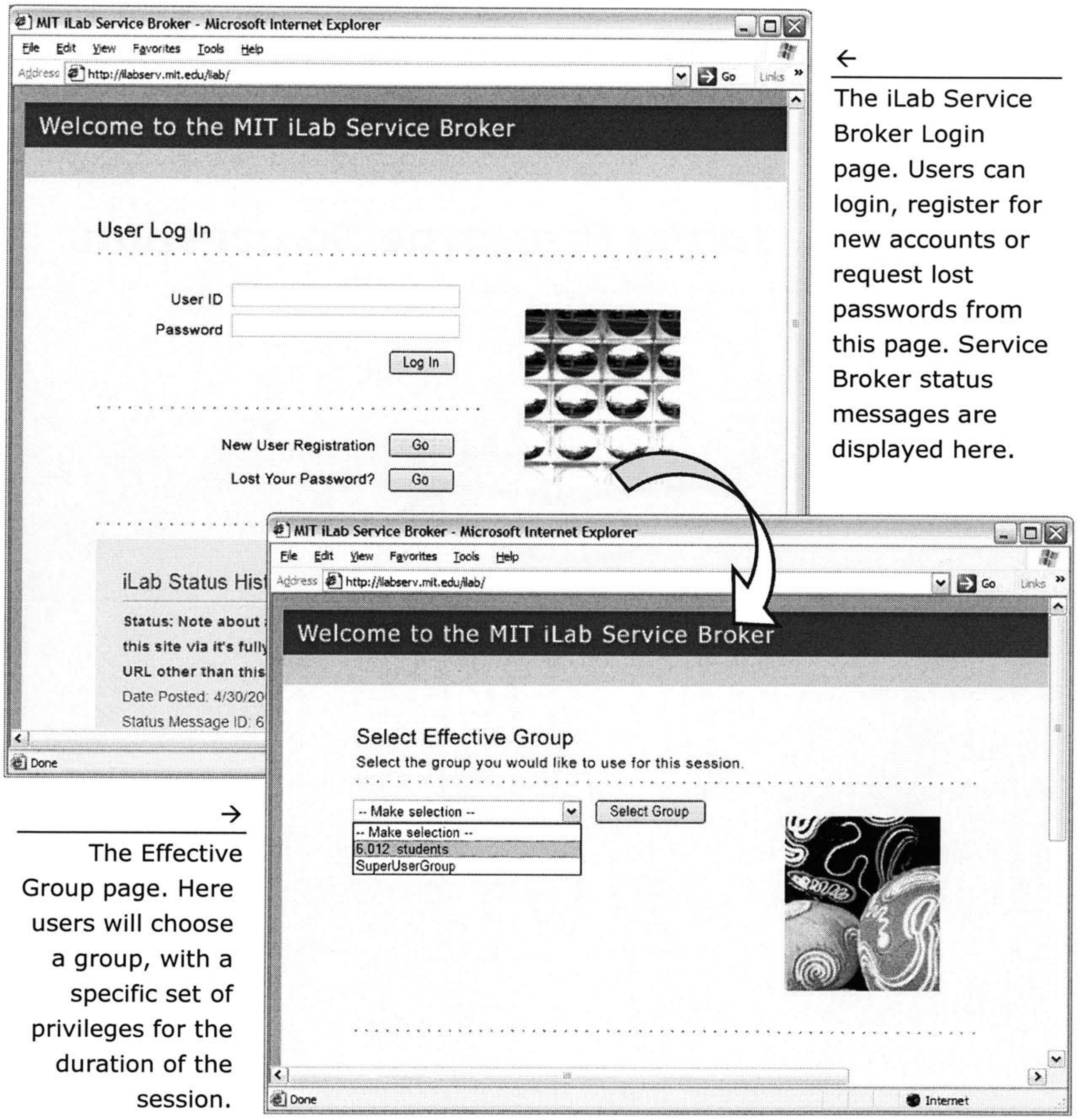
The MIT Flagpole Project consists of a number of sensors and data acquisition systems that stream different types of data to an acquisition server, where the data is published for subscribers to listen to. A parallel database server is responsible for archiving this data [3]. This project is a simple sensor lab model that involves the processing and monitoring of a number of data streams. Sensor experiments become increasingly more complicated when they allow queries to be performed over many sensors. The California Bay Area freeway is a test bed consisting of hundreds of sensors deployed by the Berkeley Highway Lab (BHL). Lab users will query these sensors to extract information like the average speed of traffic over one segment of the freeway [11].

Madden and Franklin proposed an architecture for querying multiple sensor data streams in [11]. Their architecture consists of several "sensor proxy" servers, which are intermediate workstations that serve as interfaces between deployed sensors and a "query processor" server. These sensor proxies bundle data from multiple sensors into tuples, which are routed to queries as needed. The user issues a query to a query processor that executes a plan by locating appropriate sensor proxies. It then starts the flow of tuples back to the user. Because data streams never terminate, these queries run continuously and are only removed from the system when the user explicitly ends the query [11].

General ticketing is one aspect of the project that needs to be revisited. Our general ticketing mechanism attempted to formalize the notion of secure web service invocations by placing ticket ID / passkey parameters in the headers of SOAP messages that are checked against a ticket store database. This mechanism protects various iLab entities from being accessed by clients that lack proper credentials. However, it does not protect against the possibility of a SOAP message body being intercepted and modified. WS-Security is a specification put together by a number of researchers at Microsoft, IBM and VeriSign. This specification describes a number of enhancements to SOAP messaging that enable the secure communication and integrity of SOAP messages. It specifically addresses the issue of encrypting and digitally signing the body, header and any attachments of SOAP messages [30]. The one issue with web services enhancements is that, unlike the basic web service protocols, such as XML, SOAP, and WSDL, there are few toolkits that implement them. These toolkits, such as Microsoft's WSE, when used to author web services, may actually restrict their interoperability [31]. Nevertheless, given that security is becoming a crucial requirement of web services, we expect WS-Security to become a more integrated part of the next generation web services protocols.

The next wave of design and implementation will help bring about a better understanding of interactive and sensor online labs, and the mechanisms that make it easier for these labs to integrate the iLab architecture. By tackling issues such as WS-Security, and refining the common services, the architecture will also offer a more suitable administrative and management system for online labs.

Appendix A. Initial Prototype Screenshots


MIT iLab Service Broker - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://ilabserv.mit.edu/ilab/
Go Links
Log In
Go
Go
Done
Select Group
Internet

Welcome to the MIT iLab Service Broker

User Log In

User ID
Password

New User Registration
Lost Your Password?

iLab Status Hist

Status: Note about
this site via it's full
URL other than this
Date Posted: 4/30/20
Status Message ID: 6

Welcome to the MIT iLab Service Broker

Select Effective Group

Select the group you would like to use for this session.

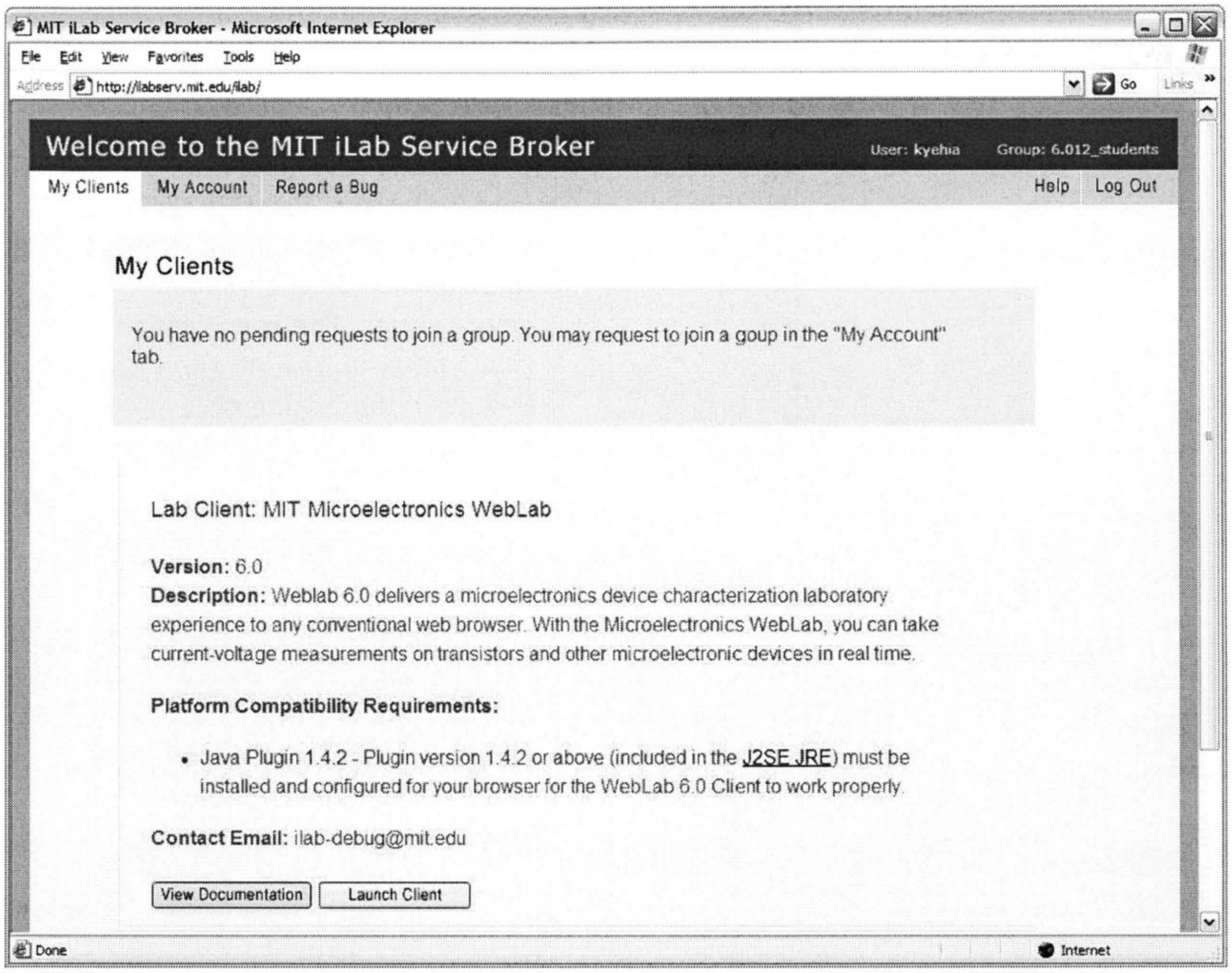
-- Make selection --
-- Make selection --
6.012_students
SuperUserGroup

←

The iLab Service Broker Login page. Users can login, register for new accounts or request lost passwords from this page. Service Broker status messages are displayed here.

→

The Effective Group page. Here users will choose a group, with a specific set of privileges for the duration of the session.


MIT iLab Service Broker - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://ilabserv.mit.edu/ilab/
Go Links
Welcome to the MIT iLab Service Broker
User: kyehia
Group: 6.012_students
My Clients
My Account
Report a Bug
Help
Log Out
View Documentation
Launch Client
Done
Internet

My Clients

You have no pending requests to join a group. You may request to join a goup in the "My Account" tab.

Lab Client: MIT Microelectronics WebLab

Version: 6.0
Description: Weblab 6.0 delivers a microelectronics device characterization laboratory experience to any conventional web browser. With the Microelectronics WebLab, you can take current-voltage measurements on transistors and other microelectronic devices in real time.

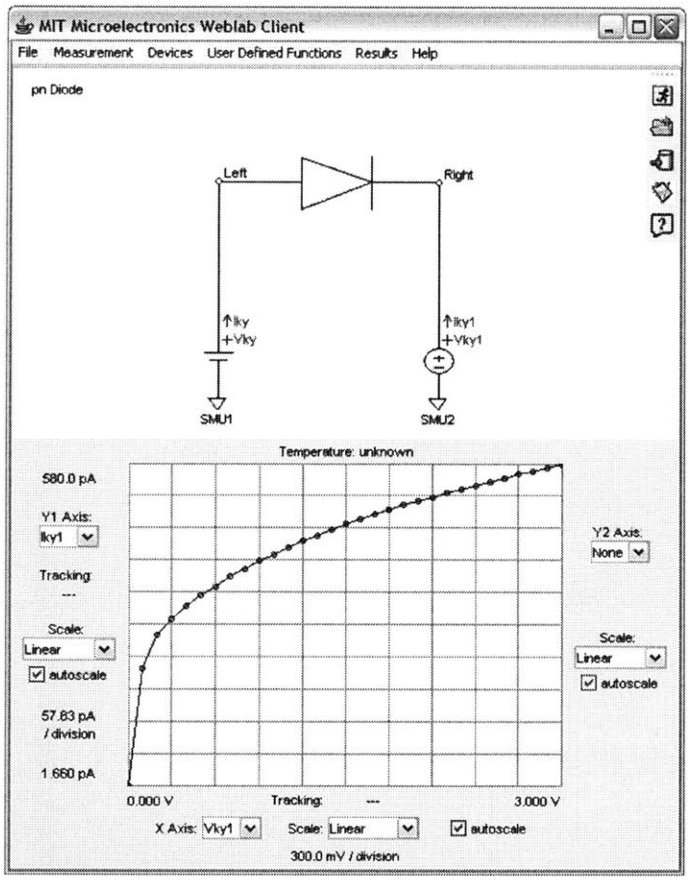
Platform Compatibility Requirements:

- Java Plugin 1.4.2 - Plugin version 1.4.2 or above (included in the J2SE JRE) must be installed and configured for your browser for the WebLab 6.0 Client to work properly.

Contact Email: ilab-debug@mit.edu

↑

In selecting a group that is not an administrator group, the user is redirected to the My Clients page. The group that was chosen above is the "6.012 students group". Here, any group-specific messages are displayed for the user in the top box. The bottom box enumerates all the clients the user can access. For each client, a description, a "launch client" button and a "view documentation" button is provided.

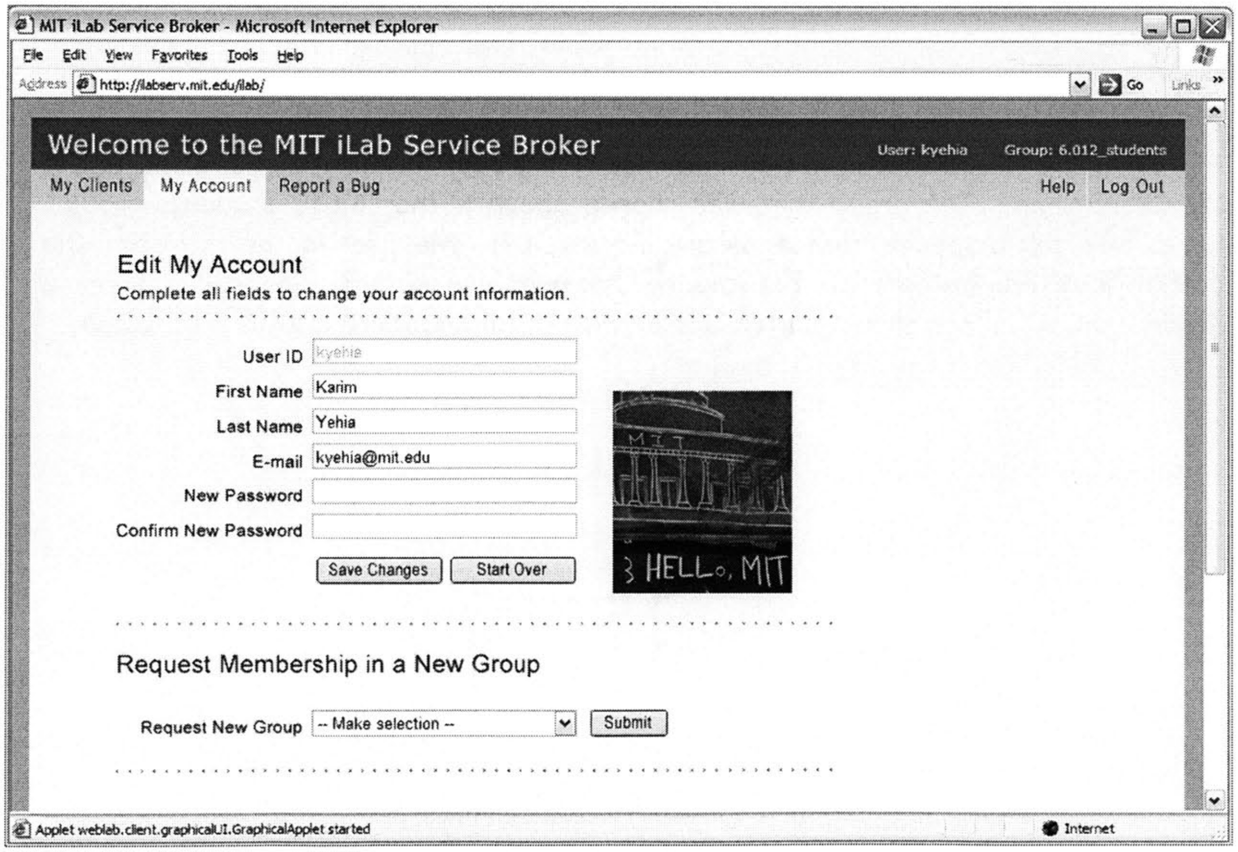


←

After clicking the "launch client" button in the My Clients page, the WebLab Java applet is launched. Users use this client to specify voltages and currents to be applied on the various device nodes. The specification is submitted to the lab server, and when the results are returned, they are displayed as a chart.

↓

Users can navigate to the My Account page from the My Clients page. Here they can modify account information and request to be placed in new groups.


MIT iLab Service Broker - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://ilabserv.mit.edu/ilab/
Go
Links
Welcome to the MIT iLab Service Broker
User: kyehia
Group: 6.012_students
My Clients
My Account
Report a Bug
Help
Log Out
Save Changes
Start Over
Request New Group
-- Make selection --
Submit
3 HELLo, MIT
Applet weblab.client.graphicalUI.GraphicalApplet started
Internet

Edit My Account

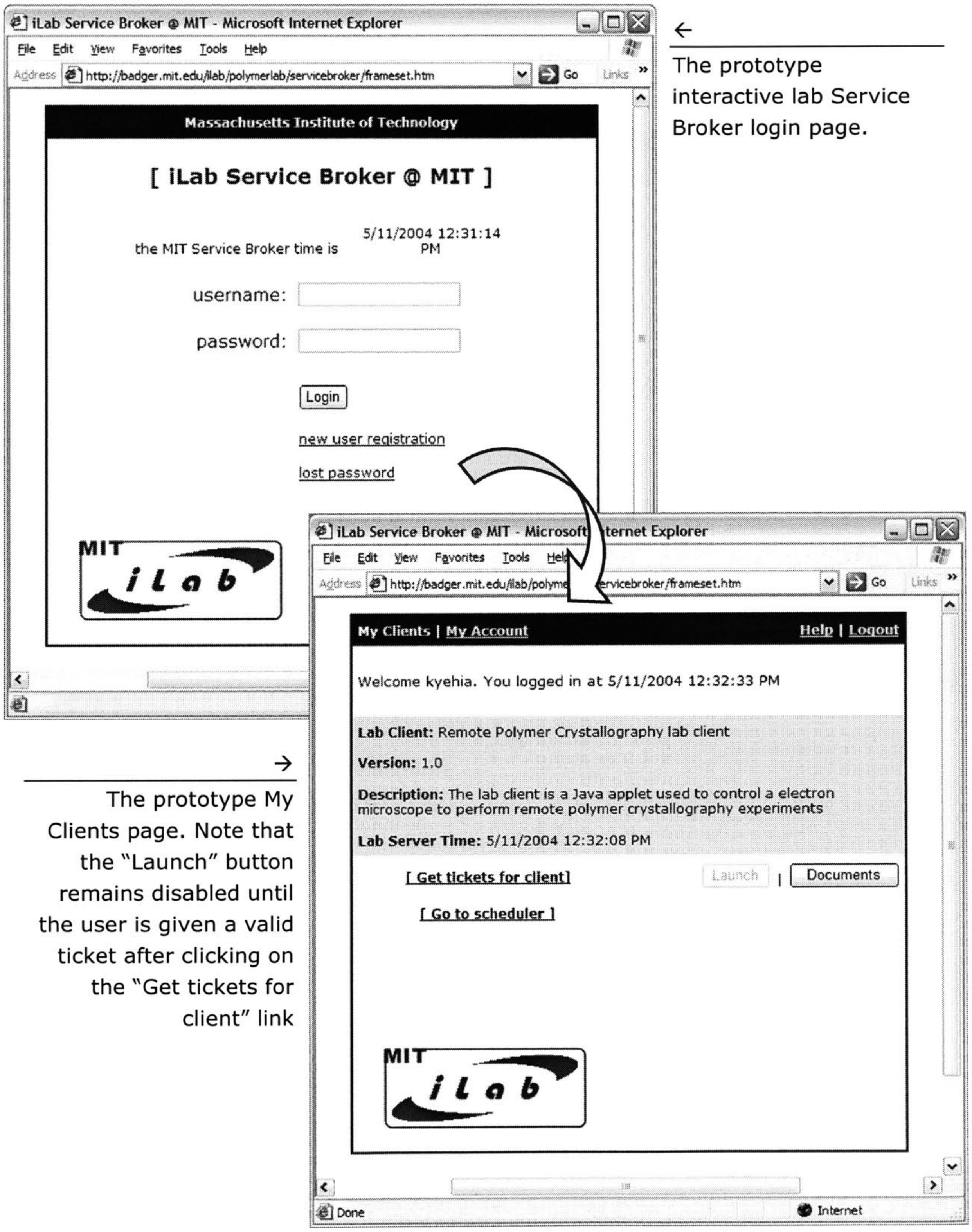
Complete all fields to change your account information.

User ID	kyehia
First Name	Karim
Last Name	Yehia
E-mail	kyehia@mit.edu
New Password	
Confirm New Password	

Request Membership in a New Group

Appendix B. The iLab Interactive Experiment

Appendix B-I. Interactive Experiment Screenshots


iLab Service Broker @ MIT - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://badger.mit.edu/ilab/polymerlab/servicebroker/frameset.htm
Go
Links
Massachusetts Institute of Technology
[iLab Service Broker @ MIT]
the MIT Service Broker time is 5/11/2004 12:31:14 PM
username:
password:
Login
new user registration
lost password
MIT iLab
My Clients | My Account
Help | Logout
[Get tickets for client]
Launch
Documents
[Go to scheduler]
Done
Internet

Welcome kyehia. You logged in at 5/11/2004 12:32:33 PM

Lab Client: Remote Polymer Crystallography lab client

Version: 1.0

Description: The lab client is a Java applet used to control a electron microscope to perform remote polymer crystallography experiments

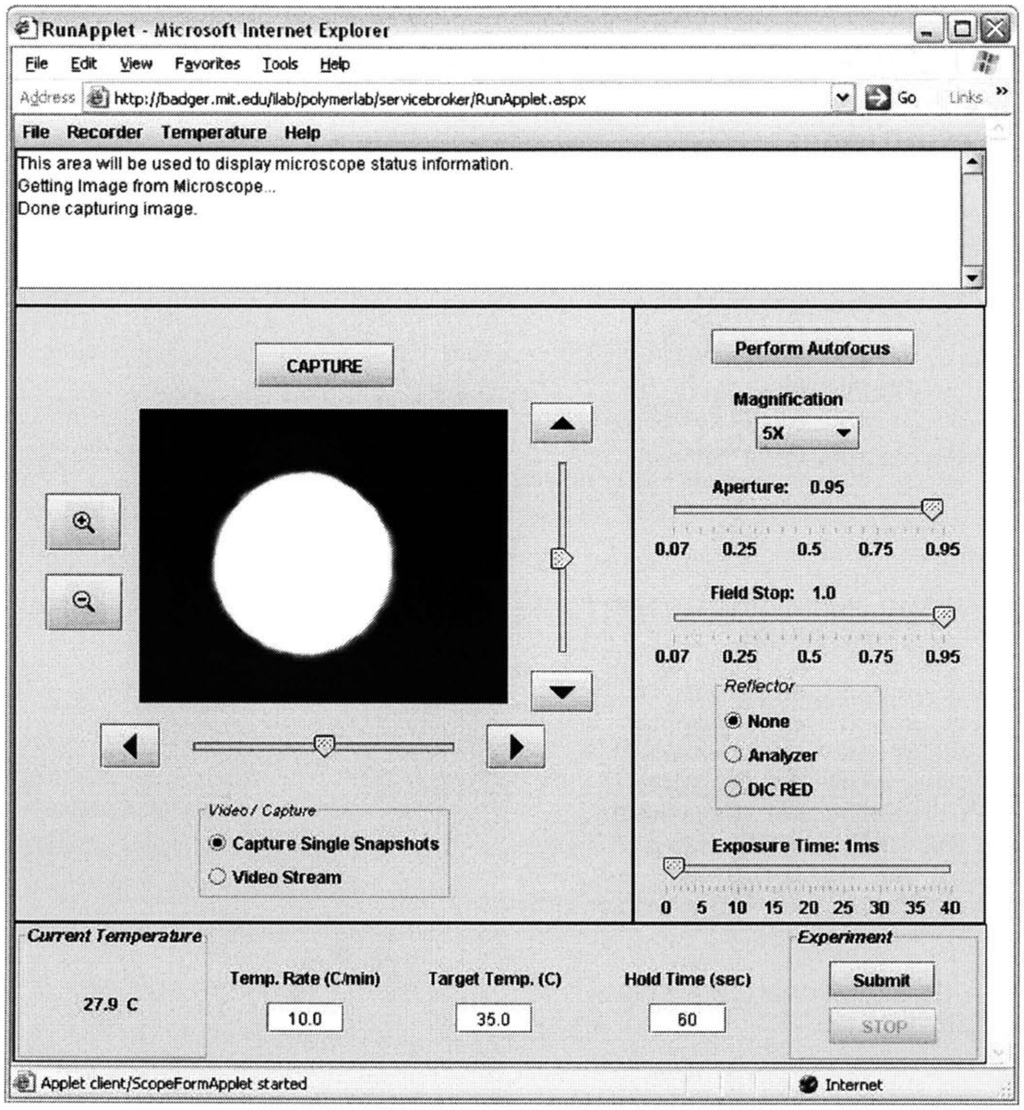
Lab Server Time: 5/11/2004 12:32:08 PM

←

The prototype interactive lab Service Broker login page.

→

The prototype My Clients page. Note that the "Launch" button remains disabled until the user is given a valid ticket after clicking on the "Get tickets for client" link



RunApplet - Microsoft Internet Explorer

http://badger.mit.edu/ilab/polymerlab/servicebroker/RunApplet.aspx

This area will be used to display microscope status information.
Getting Image from Microscope...
Done capturing image.

Magnification: 5X

Aperture: 0.95

Field Stop: 1.0

Reflector: [✓ None] Analyzer DIC RED

Exposure Time: 1ms

Video / Capture: [✓ Capture Single Snapshots] Video Stream

Current Temperature: 27.9 C

Temp. Rate (C/min): 10.0

Target Temp. (C): 35.0

Hold Time (sec): 60

Applet client/ScopeFormApplet started

↑

With a valid ticket, the "Launch" button becomes enabled, and the user can start the Polymer Crystallography Java applet which takes the ticket parameters and passes them to the lab server to initiate a direct connection.

Appendix B-II. Prototype Structs and Web Services

Object Definitions:

```
// Represents a "user" registered with the Service Broker.
public struct User
{
      public string userID;
      public string firstName;
      public string lastName;
      public string email;
      public string affiliation;
      public string password;
      public DateTime registrationDate;

}

// Represents a "service provider" registered with the Service Broker.
public struct iLabEntity
{
      public string entityID;
      public string entityType;
      public string entityName;
      public string description;
      public string webServiceURL;
      public string contactEmail;
      public string infoURL;
      public string webApplicationURL;
}

// Represents a generalized ticket
public class Ticket
{
      public int ticketID;
      public string passkey;
      public string issuerID;
      public string sponsorID;
      public string redeemerID;
      public string payload;
      public DateTime expiration;
}

// Represents a role held by a Service Provider to administer tickets for
itself and other Service Providers.
public struct RoleSet
{
      public string roleEntityID;
      public string roleType;

      //Role Types
      public const string createType = "Create";
      public const string createAndGiveType = "CreateAndGive";
      public const string cancelType = "Cancel";
}
```

Service Broker Web Service:

- *[WebMethod (Description="Method used to register with the Service Broker.")]*
  [SoapHeader("sbHeader", Direction=SoapHeaderDirection.In)]

  ```
  public bool Register(string entityID, string entityName,
                          string webServiceURL, string contactEmail,
                          string description, string webApplicationURL,
                          string infoURL, string entityType)
  ```

- *[WebMethod (Description="Method used to retrieve all entities / Service Providers registered with the Service Broker.")]*
  [SoapHeader("sbHeader", Direction=SoapHeaderDirection.In)]

  ```
  public iLabEntity[] GetRegisteredEntities()
  ```

- *[WebMethod (Description="Method used to grant the entity indicated by entityID the privilege to create and / or cancel tickets on the caller Service Provider.")]*
  [SoapHeader("sbHeader", Direction=SoapHeaderDirection.In)]

  ```
  public bool GrantTicketingPermission(string role, string entityID)
  ```

- *[WebMethod (Description="Method used to remove the privilege that the entity indicated by entityID has to create and / or cancel tickets on the caller Service Provider.")]*
  [SoapHeader("sbHeader", Direction=SoapHeaderDirection.In)]

  ```
  public bool RemoveTicketingPermission(string role, string entityID)
  ```

- *[WebMethod (Description="Method used to create tickets for redeemer some entity. The entity could be the caller of the method or it could be some other entity that the caller has been given the privilege to create tickets for.")]*
  [SoapHeader("sbHeader", Direction=SoapHeaderDirection.In)]

  ```
  public Ticket CreateTicket(string redeemerID, DateTime expiration,
              string payload, string holderID, string holderType)
  ```

- *[WebMethod (Description="Method used to create tickets for some redeemer entity. The ticket is first generated, and is then installed at the entity indicated by redeemer entity ID.")]*
  [SoapHeader("sbHeader", Direction=SoapHeaderDirection.In)]

  ```
  public int CreateAndGiveTicket(string redeemerID, DateTime expiration,
              string payload, string holderID, string holderType)
  ```

- *[WebMethod (Description="Method used to cancel the ticket indicated by ticket ID. A CancelTicket method on the sponsor of the ticket is invoked. If the ticket has previously been verified , a CancelTicket method on the redeemer is also invoked.")]*
  [SoapHeader("sbHeader", Direction=SoapHeaderDirection.In)]

  public bool **CancelTicket**(int **ticketID**)

- *[WebMethod (Description="Method returns ticket struct pertaining to ticketID if the {ticketID, passkey} combo are valid. If not valid, it returns and empty ticket.")]*
  [SoapHeader("sbHeader", Direction=SoapHeaderDirection.In)]

  public Ticket **VerifyTicket**(int **ticketID**, string **passkey**)

Lab Server Web Service:

- *[WebMethod (Description="Returns the current time on the Lab Server")]*

  public DateTime **GetCurrentTime**()

- *[WebMethod (Description="CancelTicket can be invoked by a Service Broker to tell the Lab Server that the Ticket with id ticket_id has been cancelled. Returns true if operation completes.")]*
  [SoapHeader("gtHeader", Direction=SoapHeaderDirection.In)]

  public bool **CancelTicket**(int **ticket_id**)

- *[WebMethod (Description="Use this method to install a Service Broker ticket.")]*
  [SoapHeader("gtHeader", Direction=SoapHeaderDirection.In)]

  public void **InstallTicket**(ServiceBrokerService.Ticket **t**)

- *[WebMethod (Description="Method is called by lab client that inserts ticketID / passkey in header. Returns a string password, which applet will transmit to Python server on open socket.")]*
  [SoapHeader("gtHeader", Direction=SoapHeaderDirection.In)]

  public string **Connect**()

References

[1] Aktan, B., Bohus, C.A., Crowl, L.A., Shor M.H. "Distance learning applied to control engineering laboratories". IEEE Transactions on Education (vol. 39 no. 3). August 1996.

[2] Rubaiyat, A.K. "Software architecture for web-accessible heat exchanger experiment". S.M Thesis, Dept. of Civil and Environmental Engineering. Massachusetts Institute of Technology. 2002.

[3] Amaratunga, K., Sudarshan, R. "A virtual laboratory for real-time monitoring of civil engineering infrastructure." International Conference on Engineering Education, Manchester, U.K. August 2002.

[4] Dabbagh, N. "Web-based course management systems". Education and Technology: an Encyclopedia. ABC-CLIO: Santa Barbara, CA. 2004 edition.

[5] Dawson, K. "Active learning". Education and Technology: an Encyclopedia. ABC-CLIO: Santa Barbara, CA. 2004 edition.

[6] Chang, V. "Remote collaboration in WebLab". M.Eng Thesis, Dept. of Electrical Engineering and Computer Science. Massachusetts Institute of Technology. May 2001.

[7] Harward, V.J. "The challenge of building Internet-accessible labs". iLab White paper. Center for Educational Computing Initiatives. Massachusetts Institute of Technology. November 2003.

[8] Seetharaman, K. "The CORBA connection". Communications of the ACM (vol. 41 no. 10). October 1998.

[9] Gisolfi, D. "Web services architect: Is web services the reincarnation of CORBA?". Available http: www-106.ibm.com/developerworks/webservices/library/ws-arc3/

[10] Liberty, J., Hurwitz, D. "Programming ASP.NET". O'Reilly and Associates: Sebastopol, CA. 2002 edition.

[11] Madden, S., Franklin M.J. "Fjording the stream: An architecture for queries over streaming sensor data". Available http: http://citeseer.nj.nec.com/cache/papers/cs/29679/http:zSzzSzwww.cs.berkeley.eduzSz~franklinzSzPaperszSzfjordsicde02.pdf/madden02fjording.pdf

[12] Sumra, R., Arulazi, D. "QoS for web services - demystification, limitations, and best practices". Available http: http://www.developer.com/services/article.php/2027911

[13] Gordon, A. D., Pucella, R. "Validating a web service security abstraction by typing". ACM Workshop on XML security. November 2002.

[14] Thangarathinam, T. ".NET remoting versus web services". Available http: http://www.developer.com/net/net/article.php/2201701

[15] Harward, V.J. "Service Broker to lab server API". iLab Specification paper, Center for Educational Computing Initiatives. Massachusetts Institute of Technology. October 2003.

[16] Harward, V.J., Zych, D. "Client to Service Broker API". iLab Specification paper, Center for Educational Computing Initiatives. Massachusetts Institute of Technology. September 2003.

[17] Harward, V.J., Zych, D. "Service Broker administrative API". iLab Specification paper, Center for Educational Computing Initiatives. Massachusetts Institute of Technology. October 2003.

[18] Meier, J.D., Mackman A., Dunner M., Vasireddy, S. "Building secure Microsoft ASP.NET applications: authentication, authorization and secure communication". Microsoft Press: Redmond, WA. 2003 edition.

[19] Harward, V.J. "Service Broker authentication API". iLab Specification paper, Center for Educational Computing Initiatives. Massachusetts Institute of Technology. August 2003.

[20] "BLOB - a searchDatabase definition". Available http: http://searchdatabase.techtarget.com/sDefinition/0,,sid13_gci213817,00.html

[21] "Writing BLOB values to a database". Available http: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconwritingblobvaluestodatabase.asp

[22] Harward, V.J., Choudhary, V.S. "Service Broker experiment storage API". iLab Specification paper, Center for Educational Computing Initiatives. Massachusetts Institute of Technology. August 2003.

[23] "The OKI project". Available http: http://sourceforge.net/projects/okiproject/

[24] "The Authorization OSID". Available http: http://umn.dl.sourceforge.net/sourceforge/okiproject/OSID_Authorization_rel_0_3.pdf

[25] Harward, V.J. "Service Broker authorization API". iLab Specification paper, Center for Educational Computing Initiatives. Massachusetts Institute of Technology. November 2003.

[26] Delaney, K. "Inside SQL server 2000." Microsoft Press: Redmond, WA. 2001 edition.

[27] Talavera, D.J. "Online laboratory for remote polymer crystallization experiments using optical microscopy." M.Eng. Thesis, Dept. of Electrical Engineering and Computer Science. Massachusetts Institute of Technology. May 2003.

[28] Harward, V.J. "The Experiment Storage Service API." iLab specification, Center for Educational Computing Initiatives. Massachusetts Institute of Technology. March 2004.

[29] Northridge, J.B. "A Federated Time Distribution System for Online Laboratories." S.M. Thesis, Dept. of Civil and Environmental Engineering. Massachusetts Institute of Technology. May 2004.

[30] Atkinson, B. et al. "Web services security (WS-security)." Available http: http://www-106.ibm.com/developerworks/webservices/library/ws-secure/

[31] Ewald, T. "Programming with web services enhancements 1.0 for Microsoft .NET." Available http: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwse/html/progwse.asp