# A Federated Time Distribution System for Online Laboratories

by

Jedidiah Northridge

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

© 2004 Massachusetts Institute of Technology. All rights reserved.

Author .....
Department of Civil and Environmental Engineering
May 19, 2004

Certified by .....
Steven Lerman
Professor of Civil and Environmental Engineering
Thesis Supervisor

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

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 07 2004
LIBRARIES

BARKER

# A Federated Time Distribution System for Online Laboratories

by

Jedidiah Northridge

Submitted to the Department of Civil and Environmental Engineering
on May 19, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science

## Abstract

The iLab project began in June of 2000 with the initial goal of creating Internet accessible laboratory experiments. After the successful implementation of several distinct laboratories, the goals of the project shifted to address the design and construction of a generalized infrastructure capable of supporting a wide variety of laboratory experiments. Early experiences suggested the need for a configurable experiment scheduling system. Such a system would be particularly important in the face of expected growth: as the number of providers and consumers grew, it would become crucial to empower providers with the ability to enforce experiment usage policies, to guarantee timely lab access to clients, and to maximize resource usage whenever possible.

We will explore how the present iLab infrastructure can be modified to allow for experiment scheduling. This system would be designed in keeping with two key principles: generality and architectural consistency. It would have to support disparate scheduling algorithms of varying complexity and remain faithful to the theme and priorities of the existing iLab infrastructure. Design will be based on requirements gathering and the analysis of existing remotely available experiments. Resulting changes to the iLab infrastructure will be enumerated, justified, and their ramifications discussed. This design will be implemented and considered in the same fashion. Finally, future scheduling work within the context of iLab will be described.

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

# Acknowledgments

Thanks are due to the members of the iLab and Microelectronics development teams. I am especially grateful for your willingness to stay later than expected while debating the merits of my design.

Karim Yehia built the Service Broker discussed in Chapter 4. Our lab implementation would not have succeeded without his dedicated effort and skill.

Jud Harward was a source of insight and ongoing support. His suggestions and design work played an integral role in both the General Ticketing mechanism and the Scheduling Server.

Steven Lerman provided the direction and advice that formed the basis of this thesis and guided it to completion.

Most importantly, thanks are due to my family: Michael, David, Pam, Lori, and John. Your support made this possible.

Finally, I wish to acknowledge the intangible contribution of Laura Costello, to whom this thesis is dedicated.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# iLab: A Framework for Online Laboratory Experiments

The iLab project began in June of 2000 as an iCampus research project.[1] The initial goal of the project was to create Internet accessible laboratory experiments. This goal was accomplished when several online experiments were successfully implemented at MIT. These diverse projects included the Heat Exchanger Lab, Polymer Recrystallization Lab, and Microelectronics WebLab.[2] After achieving its first goal, the focus of the project shifted to address the design and construction of a generalized online experiment infrastructure. This infrastructure would concentrate on providing solutions to problems common to all online experiments. This would enable experiment implementers to concentrate solely on the domain specific challenges of bringing their laboratory online.

The first version of the software infrastructure debuted in January of 2004 with full support for a single type of experiment known as a "Batched Experiment." A Batched Experiment allows users to submit a list of instructions for the lab to execute and then view the results as soon as the experiment has completed. In addition to supporting this experiment type, the iLab infrastructure design provided a strong sep-

[1] iCampus was initiated in October 1999 as a five-year, $25 million research alliance between MIT and Microsoft Research. For more information, please visit http://icampus.mit.edu

[2] You can visit these projects on the web at http://heatex.mit.edu, http://polymerlab.mit.edu, and http://ilabserv.mit.edu, respectively

aration between reusable, administrative components and laboratory specific components. This division enables lab implementers to take full advantage of iLab provided solutions for common tasks such as user authentication and result storage, while creating a well-defined interface that implementers can use to plug-in specific laboratory functionality.

In addition to an infrastructure design, the iLab team delivered a reference implementation of a laboratory experiment using their infrastructure.[3] This was an important first step for iLab, as it immediately illustrated the usefulness of the infrastructure and highlighted a lab that fit the Batched Experiment paradigm. However, the iLab architects recognized that not all laboratories followed the Batched Experiment model. They understood that their infrastructure would need to support experiments that allowed for continuous user interaction. Modifications to support an interactive experiment would be substantial: communication protocols would need to be rewritten, laboratory interfaces would have to be changed, and latency concerns would become significant. Additionally, shifting from a Batched Experiment to an Interactive Experiment would be accompanied with a shift in the standard usage scenario. In order to provide a student meaningful access to the lab, an experiment scheduling system would have to be created.

The goal of this thesis is to describe such a system. This system will be created by making two major modifications to the current infrastructure. The first involves re-writing the iLab communication mechanism. In the iLab infrastructure, exchanging messages with an online laboratory is tantamount to running an experiment. At present, the privilege of directly speaking to an online experiment is denied to students. This design proposes to free communication paths so that anyone with the appropriate credentials can speak directly to a laboratory. Further these credentials will be valid only for limited periods of time. Once acquired, students will only be able to run experiments during this period. This change will translate the problem of experiment scheduling into an issue of requesting and creating time-based credentials for would-be lab users.

[3] http://ilabserv.mit.edu/ilab

The second change is the creation of a new iLab entity whose sole function is to provide scheduling expertise to the iLab community. This entity will be known as the Scheduling Server, and it will accept input from lab owners, teachers, and students. It will use this input to distribute time among students by creating credentials that grant time-specific access to laboratory experiments.

This design will modify the present infrastructure so that it is able of support an extensible, flexible mechanism for experiment scheduling. To corroborate these claims, we will modify an existing interactive experiment, meet their scheduling needs, and bring the experiment into the new iLab framework.

This chapter will continue to introduce the current iLab infrastructure. It will provide the reader with a working understanding of the Batched Experiment version of the iLab software architecture, and a high level description of the suggested changes.

Chapter Two will describe the design of the new communication mechanism. This mechanism will be referred to as General Ticketing, and it will become the foundation of all communication within the iLab infrastructure. We will focus specifically on how General Ticketing can be used to enable a student to access a laboratory during a specified time period.

Chapter Three will describe the design of the Scheduling Server. In addition to making itself available to accept input from lab owners, class teachers, and students, it will support one or more scheduling algorithms, and understand how to operate within the General Ticketing environment. It will ultimately be responsible for the distribution of time and granting lab access.

Chapter Four will describe an implementation of the General Ticketing mechanism and a Scheduling Server. The Polymer Crystallization Laboratory is an interactive laboratory that allows students access to a polymer sample, a heating tray, and a polarized light microscope. Students heat the polymer sample until it melts, and then observe crystal growth as it cools. We will create a Scheduling Server that can support a first come, first served reservation system for this lab. The details of the implementation will set the stage for an analysis of the proposed design.

Chapter Five will feature an analysis of the design and associated implementation.

We will discuss any strengths and weaknesses that are uncovered. We will conclude with suggestions for future work, reflecting both the theoretical nature of the design and the practical nature of the implementation. In addition to covering the General Ticketing mechanism and Scheduling Server, we will also discuss the evolution of scheduling algorithms as the iLab infrastructure grows.

## 1.1 Status Quo

The design proposed by this thesis uses the present iLab infrastructure as a basis for developing changes. This version of the infrastructure debuted in January of 2004 and is scheduled to be released publicly in July of 2004. In order to put the suggested design changes into context, we must begin by obtaining an understanding of the current infrastructure. To this end, we will supply an overview of the infrastructure that will serve as a sufficient basis for understanding the remainder of this document. We refer readers interested in greater detail to the iLab Application Programming Interface documents [1, 2, 3].

The present iLab infrastructure can support Batched Experiments, where a Batched Experiment is defined as an experiment, "that can be submitted as a single specification and then run without further interaction with the user."[2]

There are two powerful, stand-alone entities in the iLab framework, and they are known as the Lab Server and the Service Broker. There is an additional tool that a student will directly interact with known as the Lab Client. Together these three components enable a student to run an experiment.

**Lab Server:** A Lab Server represents the laboratory experiment. It is a Web Service that enables a Service Broker to submit experiments, retrieve results, and query the status of the lab. It directly interacts with lab equipment.

Only a Service Broker can communicate with a Lab Server, and both must participate in a registration process before this can happen. The relationship between Lab Servers to Service Brokers is many to many.

**Service Broker:** A Service Broker is the entity responsible for authenticating students and enabling them to access Lab Servers. A Service Broker is both a Web Application and a Web Service.

As a Web Application, the Service Broker provides a means for students to authenticate themselves and indicate their interest in running experiments.

As a Web Service, a Service Broker hosts methods that enable users to submit experiments, retrieve results, and query the status of the lab. These methods are identical to those supported by the Lab Server, and are "pass through" methods. This means that any time a Lab Client invokes one of them, the Service Broker responds by invoking the corresponding method on the Lab Server. By forcing a Lab Client to speak to a Lab Server in this manner, the Service Broker can insure appropriate control over all student to Lab Server communication. When a Lab Server trusts a Service Broker, it implicitly trusts all of the students the Service Broker authenticates.

**Lab Client** A Lab Client is a tool that allows a student to create experiment instructions and interpret experiment results. Instructions to run an experiment are known as Experiment Specifications. Requesting an experiment to be run is known as submitting an Experiment Specification.

When a Lab Server has finished running an Experiment Specification, it will return a domain-specific Experiment Result. Lab Clients are responsible for interpreting this result for a student.

Using the knowledge of these three pieces, we can view the high level steps involved when a student runs an experiment. This use case is important to understand as it will change significantly under the interactive experiment paradigm.

1. A previously registered student directs their Internet Browser to the Service Broker. Students utilize the Web Application to authenticate themselves with a username and password.

2. After successfully authenticating themselves, the student navigates the Web Application of the Service Broker to indicate that he or she wants to run an experiment. This action requires that a Lab Client be launched.

3. The student starts a Lab Client. He or she will use this tool to author and submit Experiment Specifications. When the student is ready to start an experiment, he or she will direct the Lab Client to submit an Experiment Specification. The Lab Client will submit this to the Service Broker.

4. The Service Broker receives the Experiment Specification, and, because it has previously authenticated the student, it will submit the Experiment Specification to the Lab Server.

5. The Lab Server receives the Experiment Specification, and because it has previously registered the Service Broker, it will accept the Experiment Specification. The Lab Server will enqueue the Experiment Specification and execute it as soon as possible. The Lab Server will immediately return a Submission Report indicating, among other things, the estimated completion time of the experiment.

6. When the Lab Server finishes running the corresponding experiment, it will notify the Service Broker of the completion.

7. Upon being notified of the experiment completion, the Service Broker will collect the results from the Lab Server and will save them for later access by the Lab Client.

A general comprehension of the roles played by the Service Broker and the Lab Server is sufficient to understand the motivation behind upcoming design decisions. Several details that are important to the iLab infrastructure, but not important to our proposed design have been omitted. Again, interested readers are referred to the iLab Application Programming Interface documents [1, 2, 3].

It is important to recognize what happens in the scenario where many Lab Clients submit Experiment Configurations at the same time. Each Experiment Specification

will be accepted by a Service Broker, forwarded to a Lab Server, and then enqueued at the Lab Server. With each Lab Server acceptance, the Lab Server will return an estimated time to completion. If a student feels this completion time is too long to wait for, he or she is free to suspend their iLab activity and resume it at a later time. When the Lab Server finishes running Experiment Configurations, it will notify the Service Broker. The Service Broker will save the results and make them available to the students. This means that a student can leave instructions for an experiment and be guaranteed that whenever he or she chooses to return to the Service Broker, the results, if ready, will be available for analysis.

With a Batched Experiment, the student's entire laboratory experience is contained in the creation of the Experiment Specification and the interpretation of the results. Even if the Lab Server is busy, both of these items can be saved, allowing for the laboratory experience to be revisited at the student's leisure. This is not at all the case with interactive experiments, and indicates why scheduling functionality is integral in supporting a new usage scenario.

## 1.2 Scheduling Changes

With an understanding of the present infrastructure, we can take a closer look at the changes suggested by our design. The first change affects the communication mechanism and will be known as General Ticketing. General Ticketing will enable time-restricted, credential-based communication with a Lab Server. This will become the basis on which scheduling is implemented, and a new entity known as the Scheduling Server will use General Ticketing to allow access to Lab Servers.

### 1.2.1 General Ticketing

As we have seen, only two entities types exist in the iLab infrastructure, a Service Broker and a Lab Server. After they have registered with one another, a Service Broker can invoke methods on a Lab Server, and a Lab Server can invoke methods on a Service Broker. Once this communication path has been created, it allows for

unrestricted, non-expiring communication.

Our design proposes to eliminate this static communication path, and allow any Web Service to be invoked so long as the invocation is accompanied with a set of credentials. These credentials will be known as *Tickets* and will be used to identify the invoker and their associated permissions. Tickets will accompany all Web Service invocations, but will enable scheduling functionality when they accompany Lab Server invocations. In particular, Lab Server Tickets will contain information indicating a time period when the Ticket is valid. For example, a Ticket could be created that would grant access to a Lab Server and it might be valid between 2 and 3PM EST on April 19th, 2020. Under General Ticketing, such a Ticket could be created months in advance or just seconds before the Lab Server was to be accessed. As long as the Ticket is *valid*, the Lab Server will grant access to the holder. Greater details on General Ticketing will be provided in Chapter 3, including how Tickets are created and validated.

### 1.2.2 Scheduling Server

Without introducing any new entities, either the Lab Server or the Service Broker must be responsible for creating and distributing Tickets. However, both of these choices force each entity to take on functionality outside of its boundaries. Instead, we claim that the creation of a new entity, the Scheduling Server, is in keeping with the architecture to date and an efficient way to proceed.

A Scheduling Server will be an optional component of the iLab framework, and it will specialize in using scheduling algorithms to distribute time on Lab Servers among lab clients. It will have the ability to create Tickets that lab clients can use to access a Lab Server. A Scheduling Server will provide its functionality both as a Web Service and a Web Application. It will use Web Services to support Ticket management and distribution.

As a Web Application, it will allow administrators of Lab Servers along with students and teachers from Service Brokers the ability to provide input to the scheduling process. For example, a Lab Server Administrator could indicate when the Lab would

be available. A teacher could indicate rules that their students must follow when signing up for time. And, finally, a student could indicate times that fit best into their schedule.

At a minimum, a Scheduling Server will implement a single scheme for distributing time. The details of how people interact with the Scheduling Server will be decided by its implementers. It will tie into the iLab framework by issuing Tickets that are in keeping with the General Ticketing mechanism. As such, the Scheduling Server enjoys a strong layer of abstraction. It can choose to implement any scheduling algorithm it deems necessary in order to create Tickets, and this is particularly important, because, while present scheduling needs are simple, we expect them to greatly grow in complexity as more iLabs come into existence.

## 1.3 Scheduling Implementation

To test our proposed scheduling design, we will implement it for an interactive experiment. We will take an existing, functioning lab and bring it into the iLab framework. This lab is known as the Polymer Crystallization Lab, and it initially came into existence during the Spring of 2002. It provides remote, real-time access to a microscope, a heating stage, and polymer samples. It was most recently used during the Fall of 2003 to teach the class 10.467: *Polymer Science Laboratory* at MIT.

In completing this implementation, we hope to ascertain the strengths and weaknesses of the proposed design. Finally, we'll analyze these aspects, and propose future work in three distinct areas: the General Ticketing mechanism, as it relates to scheduling, the implementation of the Scheduling Server, and general scheduling algorithms in the iLab Framework.

# Chapter 2

# General Ticketing

iLab experience to date suggests that operators of Lab Servers are not necessarily interested in the details of individual students. Instead, a Lab Server is willing to trust a Service Broker with the task of student authentication. In the present version of the iLab infrastructure this relationship is enforced by protocol: Lab Servers will not respond to requests made by students.

General Ticketing supports a distributed authentication model in a different manner. Under General Ticketing all communication must be accompanied with a set of credentials. These credentials, known as Tickets, are created at the request of iLab Entities, such as a Service Broker or Lab Server.[1] A Ticket simply represents a resource. They are created often and generally have a short lifespan. A Service Broker could create and give a Ticket to a student that would enable them to communicate directly with a Lab Server.

Tickets enable all communication within the iLab infrastructure. However, we will concentrate on how they form the foundation for providing time-based access to Lab Servers, and in turn, serve as the basis for experiment scheduling. This chapter will provide a detailed look at the General Ticketing mechanism, and the next chapter will describe how the Scheduling Server will use this mechanism to schedule experiment time.

[1] A Ticket in the Kerberos authentication protocol is not related to a General Ticket. Section 2.9 discusses the similarities and differences between them.

## 2.1 Central Ticket Management

The Service Broker handles the administrative details surrounding Ticket management. This centralized structure was selected to minimize the functionality Lab Servers and Scheduling Servers would have to implement in order to support Ticketing. The burden of managing Tickets is significant; in a heavily utilized iLab system, thousands of Tickets may be created each day, and each Ticket has the potential to have an infinite lifetime. With centralized Ticket management, all iLab entities must be aware of how Tickets are used to access resources, and some iLab entities must be aware of how Tickets are created, but only the Service Broker is responsible for implementing the details supporting Ticket creation, verification, and persistence.

To better understand Ticket usage, we can examine the high-level steps involved when one iLab Entity wants to create a Ticket to be redeemed on another. Specifically, we will consider the situation when a Scheduling Server wants to create a Ticket to be redeemed on a Lab Server. We will continue to flesh out this scenario with greater detail as we progress through the chapter.

1. A Scheduling Server contacts a Service Broker with the request that a Ticket be created. It indicates that this Ticket should be redeemable on a specific Lab Server, during a certain time period.

2. The Service Broker creates the Ticket as requested, and returns it to the Scheduling Server. Even though the Ticket has been given to the Scheduling Server, the Service Broker will store all of the information associated with it. In the future, having this information will allow the Service Broker to verify the contents of the Ticket. Tickets are assigned an expiration date when they are created, and the Service Broker is responsible for storing a Ticket until it has expired. While most Tickets expire in a relatively short period of time, they have the potential to exist indefinitely.

3. Later, a student wants to access the Lab Server. He or she acquires the Ticket created in Step 2, and presents it to the Lab Server along with a request for

access.[2]

4. The Lab Server receives the Ticket along with the student's request for access. As in the previous version of the iLab infrastructure, the contents of each Web Service invocation are encrypted. However, the Lab Server may fear that the student has tampered with or forged their Ticket. It can verify the Ticket content by contacting the Service Broker.

5. The Service Broker receives a Ticket verification request from the Lab Server, and responds authoritatively with the contents of the Ticket. These contents provide the Lab Server with the information it requires to process the student's request.

In the scenario above, we say that the Scheduling Server is the *Ticket Writer* or *Sponsor*. We say that the Lab Server is the *Ticket Redeemer*. As the scenario indicates, Ticket Writers need only understand how to write Tickets and how to request that they be created. Ticket Redeemers need only comprehend Tickets and how to verify their contents. All other administrative details surrounding Tickets are handled by the Service Broker.

### 2.1.1 Service Broker Domains

We define a *Service Broker Domain* to be a collection of iLab Entities that co-operate with a single Service Broker (SB) to provide experiment access to students. Every domain contains a Service Broker, and no domain can contain multiple Service Brokers. Figure 2-1 contains two Service Broker Domains: Domain A is composed of a Service Broker, a Lab Server (LS), and a Scheduling Server (SS), and represents the three Entities from our scenario above. Domain B is composed of a Service Broker and a Lab Server. The Service Broker is the Central Ticket Manager for each domain, and the Tickets it issues can only be redeemed on entities within the domain.

With the exception of Service Brokers, iLab Entities can belong to multiple domains. It is expected that a Lab Server will provide its services to many Service

[2]We will answer the question, "How does the student acquire the Ticket?" in Chapter 3.

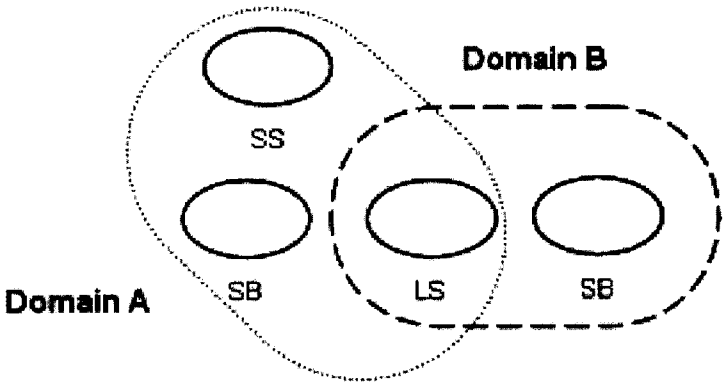


Figure 2-1: Service Broker Domains

Brokers; for example, the Lab Server in Figure 2-1 serves both Domain A and B. The concept of a Service Broker Domain allows us to define a meaningful scope for Tickets and avoid confusion. As we will learn in Section 2.2, each Ticket contains information indicating which domain it belongs to.

### 2.1.2 Entity Identifiers

The General Ticketing mechanism requires that every iLab Entity be associated with a globally unique identifier. These identifiers mirror the concepts of Lab Server and Service Broker GUIDs from the previous iLab infrastructure.[3] When implemented, an Entity ID will likely be a string of characters similar to `0921bf1379r-da5id-a9e8j7d`, but our examples will use simpler, more easily distinguished names. We assign the Service Broker, Scheduling Server, and Lab Server from Domain A in Figure 2-1 Entity IDs of `mitServiceBroker`, `mitSchedulingServer`, and `mitLabServer` for use in upcoming examples.

## 2.2 Ticket Description

A Ticket is a small collection of immutable information, and Figure 2-2 is a representation of one in the Unified Modelling Language. Only a basic understanding of

[3]See [2, p. 3] for a description of GUIDs

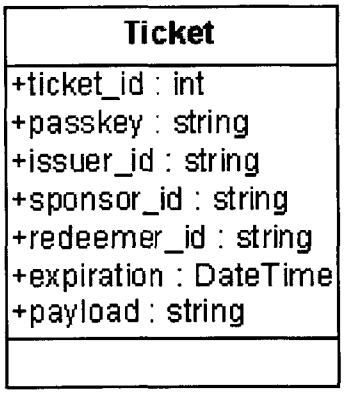


Figure 2-2: General `Ticket` (Unified Modelling Language)

the Unified Modelling Language is needed to comprehend Figure 2-2, but interested readers can read more about it at [4]. Figure 2-2 describes a Ticket as a methodless class with seven public fields.

The `issuer_id` field of a Ticket identifies the Entity ID of the Service Broker that created it. The `ticket_id` field is produced by the Service Broker and is guaranteed to be unique within its Service Broker Domain. Consequently, the combination of `ticket_id` and `issuer_id` uniquely identifies any iLab Ticket. Any Ticket created by the Service Broker in Domain A from Figure 2-1 will have `issuer_id = mitServiceBroker`

Tickets are associated with a `passkey`, and a holder must have the `ticket_id`, `issuer_id`, and `passkey` to redeem a Ticket. The `passkey` field plays a role similar to the `OutgoingServerPasskey` and `IncomingServerPassKey` from the previous version of the iLab infrastructure.[4]

Tickets can only be created at the request of iLab Entities. The `sponsor_id` field identifies the iLab Entity that requested a Ticket be created by referring to its Entity ID. A Ticket created at the request of the Server in Figure 2-1 would have `sponsor_id = mitSchedulingServer`, and we would say the MIT Scheduling Server is the *Sponsor* of the Ticket.

The `redeemer_id` field identifies the entity that the Ticket can be used at. If the Scheduling Server created a Ticket to be redeemed on the Lab Server from Figure 2-1,

[4]See [3, p. 7] more information on ServerPasskeys

```
<payload>
 <start_time>632194823532323948</start_time>
 <end_time>632194859532323948</end_time>
 <user_id>jsmith</user_id>
 <group_id>10.467</group_id>
<payload>
```

Figure 2-3: Possible Payload of Lab Server Ticket

it would have `redeemer_id = mitLabServer` and we would say the MIT Lab Server is the *Redeemer* of the Ticket.

The `expiration` field identifies a the time after which the Ticket is no longer valid. The type of the `expiration` field is `DateTime`, which uses a positive number of 100-nanosecond units beginning at midnight on January 1, 0001, Common Era to represent an instant in time. If this value is `-1`, the Ticket will never expire.

The `payload` field can contain a string of characters that indicates the purpose and privilege of a Ticket. The contents of this string are assigned at creation time by the Ticket Writer. There is no defined format for a payload, and it is possible for a Ticket Writer to create a payload that is meaningless to a Ticket Redeemer. The General Ticketing mechanism takes the approach that no single payload, or list of payloads, will be acceptable to all iLab implementers. Consequently, Ticket Writers are allowed to create Tickets with any payload, and Ticket Redeemers are free to accept or reject payloads as they see fit. We expect that implementers of a Ticket Redeemer will decide upon a list of acceptable payloads and then publish their decision to anyone who wants to use their service.

Though intentionally undefined, Figure 2-3 contains a `payload` that could potentially be used in a Lab Server Ticket. This payload is encoded in the Extensible Markup Language. The `start_time` and `end_time` elements represent the number of 100-nanosecond units that have occurred since January 1, 0001, Common Era. The other two elements identify a user and their effective group, which are meaningful in the context of a particular Service Broker Domain.

```
<?xml version="1.0" encoding="utf-8">
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
 <env:Header>
  <g:GeneralTicketHeader xmlns:g="http://ilab/GeneralTicketHeader">
   <ticket_id>100</ticket_id>
   <passkey>100Passkey</passkey>
   <issuer_id>mitServiceBroker</issuer_id>
  </GeneralTicketHeader>
 </env:Header>
 <env:Body>
     ...
 </env:Body>
</env:Envelope>
```

Figure 2-4: Soap Envelope with General Ticket Headers

### 2.2.1 Ticket Usage

Tickets are used as credentials to enable communication. They are primarily used to enable Web Service invocations. However, they can also be used to begin Web Application sessions. When Tickets are used, their `ticket_id`, `passkey`, and `issuer_id` are sent to a recipient. The recipient is then free to invoke `VerifyTicket(ticket_id, passkey)` on the Service Broker that issued the Ticket. `VerifyTicket()` returns the authoritative contents of the Ticket. All Web Service invocations in the iLab framework are protected by Secure Socket Layer encryption.

**Web Service Usage:** The iLab infrastructure invokes Web Services using the SOAP protocol in combination with HTTP. A SOAP message is composed of an Envelope, a Header, and a Body. A Ticket is associated with a Web Service invocation by including its `ticket_id`, `passkey`, and `issuer_id` in the Header of the SOAP Message. Figure 2-4 contains a SOAP Message with a Header containing Ticket information.

**HTTP Usage:** If a Web Application wants to use Tickets for authenticating users, the Tickets can be included as parameters in the Web Application's URL as in Figure 2-5

```
https://ilab/index.html?ticket_id=1&passkey=test&issuer_id=mitSB
```

Figure 2-5: Web Application URL with General Ticketing Parameters

The concepts of Web Services, SOAP, and HTTP are heavily utilized within the iLab infrastructure, and readers are assumed to be comfortable with them. Interested readers can learn more about these concepts at [8, 9, 10].

## 2.3 Application Programming Interface

In order to support the General Ticketing mechanism, changes will be made to the Web Service Application Programming Interfaces of the Service Broker and all Service Providers. The existing interfaces are described here [2, 3]. The methods in the next two sections are described in an object oriented manner. Each method can be thought of as an instance method on a singleton object whose type is one of `ServiceBroker`, `LabServer`, or `SchedulingServer`. This description style is inherited from the earlier iLab Application Programming Interfaces, and interested readers are referred to [2, 3].

### 2.3.1 Service Broker Ticket Management

The following methods are implemented by the Service Broker to support central Ticket management. They address issues concerning Ticket creation, destruction, verification, and delivery. These methods will be invoked on the Service Broker by either the Lab Server or the Scheduling Server. When any of the methods are invoked, the Header of the accompanying SOAP Message must be filled with information identifying a *Service Broker Ticket*. This type of Ticket will be described in Section 2.4, but it suffices to know that it will allow the Service Broker to determine the identity of the method invoker.

```
Ticket CreateTicket(string redeemer_id, DateTime expiration,
string payload)
```

This method can be used to create a Ticket. The resulting Ticket will have its

`ticket_id` and `passkey` generated by the Service Broker. The `issuer_id` of the Ticket will be the Entity ID of the Service Broker. The `sponsor_id` will be the Entity ID of the invoker of `CreateTicket()`. The parameters of this method determine the `redeemer_id`, `expiration`, and `payload` of the resulting Ticket. See Section 2.6 for more information.

`boolean CancelTicket(int ticket_id)`

When this method is invoked, the Service Broker will attempt to cancel the Ticket identified by `ticket_id`. If no such Ticket exists, this method will return `false`. A cancelled Ticket is no longer redeemable for resources. It is possible that a Ticket cannot be cancelled. As detailed in Section 2.3.2, each Service Provider has their own `CancelTicket()` method. When `CancelTicket()` is invoked on a Service Broker, the Service Broker will invoke `CancelTicket()` on the Redeemer of the Ticket. The Redeemer will return a `boolean` indicating if the Ticket can be cancelled. If the Redeemer returns `true` then the Service Broker will return `true`, otherwise the Service Broker will return `false`. If the Redeemer is unavailable, the Service Broker will return `false`. See Section 2.7 for more information.

`Ticket VerifyTicket(int ticket_id, string passkey)`

This method will return the authoritative version of the Ticket identified by `ticket_id` and `passkey`. If no such Ticket exists or if the `ticket_id` and `passkey` do not match, `null` will be returned.

`int CreateAndGiveTicket(string redeemer_id, DateTime expiration,`
`string payload, string recipient_id)`

This method can be used to create a Ticket and have it delivered to an iLab Entity. The iLab Entity will then use the Ticket for communicating with other iLab Entities. The Service Broker will create a Ticket using the `redeemer_id`, `expiration`, and `payload` in the same manner as `CreateTicket()` would. Next, the Service Broker will invoke `InstallTicket()` on the iLab Entity identified by `recipient_id`. The

`recipient_id` must match an Entity ID in Service Broker Domain. The Service Broker will return the `ticket_id` of the created Ticket. See Section 2.8 for more information.

`boolean GrantTicketingPermission(string role, string entity_id)`

One iLab Entity can use this method to inform the Service Broker that another iLab Entity can invoke `CreateTicket()` and `CreateAndGiveTicket()` on its behalf. This method can also be used to enable an iLab Entity the power to invoke `Cancel()` on Tickets that are redeemable on it. The `role` parameter is a `string` that must be one of `Create`, `CreateAndGive`, or `Cancel`. The role will be granted to the iLab Entity associated with `entity_id`. The `entity_id` must match an Entity ID in Service Broker Domain. For example, consider a Lab Server with Entity ID `mitLabServer` that wants to allow a Scheduling Server with Entity ID `mitSchedulingServer` the ability to create Tickets that will be redeemable on it. The `mitLabServer` can invoke `GrantTicketingPermission(Create, mitSchedulingServer)`. See Section 2.5 for more information.

`boolean RemoveTicketingPermission(string role, string entity_id)`

An iLab Entity can use this method to disallow another iLab Entity from invoking `CreateTicket()` or `CreateAndGiveTicket()` on the Service Broker, and `CancelTicket()` Tickets for it. The `entity_id` must match an Entity ID in Service Broker Domain. For example, if `mitLabServer` wanted to take the ability to create Tickets away from `mitSchedulingServer`, the `mitLabServer` can invoke `RemoveTicketingPermission(Create, mitSchedulingServer)`. See Section 2.5 for more information.

`iLabEntity[] GetiLabEntities()`

This method enables iLab Entities within a Service Broker Domain to discover other members that are in the same domain. An `iLabEntity` is a collection of information common to all iLab Entities, and is represented in the Unified Modelling

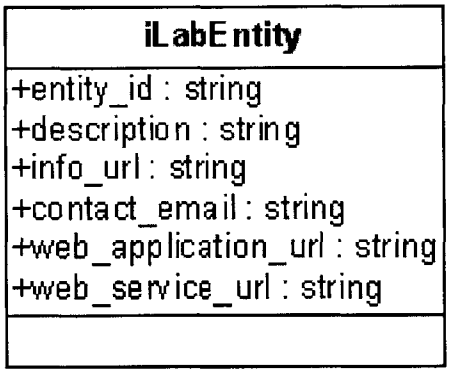


Figure 2-6: `iLabEntity` (Unified Modelling Language)

Language in Figure 2-6.

### 2.3.2 Service Provider Ticket Management

The following methods must be implemented by all Service Providers in the iLab infrastructure. At present, that includes the Lab Server and the Scheduling Server. These methods provide a mechanism for the Service Broker to deliver and destroy Tickets. When any of the methods are invoked on the Service Broker, the Header of the accompanying SOAP Message must be filled with information identifying a *Service Broker Identification Ticket.* This type of Ticket will be described in Section 2.4, but it suffices to know that it allows a Service Provider to identify the Service Broker as the method invoker.

`void InstallTicket(Ticket t)`

iLab Entities use Tickets to communicate, just as students do. For example, a Lab Server must have a Service Broker Ticket in order to invoke methods on a Service Broker. This method can be used to give a Ticket to an iLab Entity. See the definition of `CreateAndGiveTicket()` in Section 2.3.1 and the use case in Section 2.4 for more information.

`boolean CancelTicket(int ticket_id)`

A Service Broker can use this method to request that a Service Provider cancel

the Ticket identified by `ticket_id`. The Service Provider must return `true` if it can cancel the Ticket. See Section 2.7 for more information.

`DateTime GetCurrentTime()`

This method is used to return the current time according to a Service Provider. The iLab infrastructure does not demand that iLab Entities synchronize their time, and this method provides a mechanism to handle time-sensitive Tickets.

## 2.4 Initial Ticket Distribution

When a Service Broker is first brought online, it will not be aware of any other iLab Entities. This section focuses on how iLab Entities become aware of one another using General Ticketing. We will examine the process by which a Service Broker initiates communication with a Lab Server. This process can be repeated for any number of additional Lab or Scheduling Servers.

We begin by assuming a certain level of person-to-person interaction: a teacher must somehow learn that a Lab Server exists and contact the owner of it. During their initial conversation, the teacher and lab owner must exchange information equivalent to what is included in an `iLabEntity`, as seen in Figure 2-6. With this knowledge, the Service Broker creates two Tickets:

**Service Broker Ticket:** This Ticket will be used to identify the **Lab Server to the Service Broker** in all future Web Service invocations. The `ticket_id`, `passkey`, and `issuer_id` of this Ticket will be included with any invocations of `CreateTicket()`, `CancelTicket()`, or `VerifyTicket()`. The `issuer_id`, `redeemer_id`, and `sponsor_id` of this Ticket will be the Entity ID of the Service Broker.

**Service Broker Identification Ticket:** This Ticket will be used to identify the **Service Broker to the Lab Server** in all future Web Service invocations. The `ticket_id`, `passkey`, and `issuer_id` of this Ticket will be included with

| Field | Ticket1 | Ticket2 |
|---|---|---|
| ticket_id | 100 | 200 |
| passkey | 100Passkey | 200Passkey |
| issuer_id | mitServiceBroker | mitServiceBroker |
| sponsor_id | mitServiceBroker | mitServiceBroker |
| redeemer_id | mitServiceBroker | mitLabServer |

Table 2.1: Initial Ticket Values

```
ticket_id : 200
passkey : 200Passkey
issuer_id : mitServiceBroker
InstallTicket(Ticket1);
```

Figure 2-7: `InstallTicket()` Invocation

any invocations of `InstallTicket()` or `CancelTicket()`. The `issuer_id` and `sponsor_id` of this Ticket will be the Entity ID of the Service Broker. The `redeemer_id` of this Ticket will be the Entity ID of the Lab Server.

If the Entity IDs of the Service Broker and Lab Server are `mitLabServer` and `mitServiceBroker`, then the contents of these two Tickets are described in Table 2.1. We will refer to these two tickets as `Ticket1` and `Ticket2`. We omit both the `expiration` and `payload` as these items are not necessary for our present discussion. In an actual implementation, these Tickets will be easily distinguished from one another by their payloads; for now we can differentiate between them based on their `redeemer_id`. `Ticket1` is the Service Broker Ticket because its `redeemer_id` indicates that it will be used when invoking methods on the Service Broker. `Ticket1` will be included in the SOAP Header of any method the Lab Server invokes on the Service Broker. `Ticket2` is the Service Broker Identification Ticket because its `redeemer_id` indicates that it will be used when invoking methods on the Lab Server. `Ticket2` will be included in the SOAP Header of any method the Service Broker invokes on the Lab Server.

Once these Tickets have been created, the Service Broker will invoke `InstallTicket()` on the Lab Server. It will include the `ticket_id`, `passkey`, and `issuer_id` of `Ticket2` in the Header of the invocation (see Figure 2-7).

```
ticket_id : 100
passkey : 100Passkey
issuer_id : mitServiceBroker
VerifyTicket(200, 200Passkey);
```

Figure 2-8: `VerifyTicket()` Invocation

We use this notation of Figure 2-7 to denote that `InstallTicket()` was invoked, and the Header of the SOAP Message indicated `ticket_id = 200`, `passkey = 200Passkey`, and `issuer_id = mitServiceBroker`.

When `InstallTicket()` is invoked on Lab Server, it examines the Header and recognizes that it has never seen a Ticket with `issuer_id = mitServiceBroker` before. In this special case, it assumes that the Header information identifies a Service Broker Identification Ticket and the parameter of `InstallTicket(Ticket1)` is a Service Broker Ticket that it should use for future communication with `mitServiceBroker`. Further, from the previous person-to-person conversation, it can associate the `issuer_id` with a target for Web Service invocations. The next step is for the Lab Server to confirm that the Ticket included in the Header of `InstallTicket()` is legitimate. It does this by invoking `VerifyTicket()` on the Service Broker (see Figure 2-8).

When `VerifyTicket()` is invoked as in Figure 2-8, the Service Broker can examine the Header information and identify that Ticket it gave exclusively to `mitLabServer` is being used. It recognizes the Ticket and returns a full copy of `Ticket1`. If it did not recognize the Ticket, it would return `null`. This scenario can be repeated for additional Lab or Scheduling Servers.

## 2.5 Ticket Creation Permissions

When an iLab Entity first joins a Service Broker Domain it is only allowed to create Tickets that are redeemable on it.[5] However, it may wish to allow another iLab Entity to create Tickets on its behalf. Any iLab Entity can invoke `GetiLabEntities()`

[5]This even holds true for the Service Broker. Even though it is the Central Ticket Manager, iLab Entities must explicitly grant the Service Broker Ticket creation permissions.

```
ticket_id : 100
passkey : "100Passkey"
issuer_id : mitServiceBroker
GrantTicketingPermission("Create", mitSchedulingServer);
```

Figure 2-9: `GrantTicketingPermission()` Invocation

```
ticket_id : 300
passkey : "300Passkey"
issuer_id : mitServiceBroker
CreateTicket(mitLabServer, exp, pay);
```

Figure 2-10: `CreateTicket()` Invocation

on the Service Broker to query all of the other iLab Entities that are in the Service Broker Domain. In particular, the iLab Entity now knows all the Entity IDs of every entity in the domain. With this information, an iLab Entity can use `GrantTicketingPermission()` to inform the Service Broker that another iLab Entity should be able to manage its Tickets. For example, imagine a Lab Server with Entity ID `mitLabServer` wants to allow a Scheduling Server with Entity ID `mitSchedulingServer` to invoke `CreateTicket()` its behalf. To accomplish this, the Lab Server invokes `GrantTicketingPermission()` as in Figure 2-9. After the method is invoked the Service Broker will allow the Scheduling Server to invoke `CreateTicket()` and specify that `redeemer_id = mitLabServer`.

## 2.6 Ticket Creation

At the beginning of this Chapter, we described a scenario in which a Scheduling Server was interested in creating a Ticket to be redeemed on a Lab Server. If iLab implementers followed the steps in Sections 2.4 and 2.5, this could be accomplished with a single method invocation, as seen in Figure 2-10. This invocation assumes that Service Broker Ticket held by the Scheduling Server is described in Table 2.2.

In Figure 2-10, `exp` is a `DateTime` indicating the expiration of the Ticket, and `pay` is a `string` described in Figure 2-3.

| Field | Ticket3 |
|---|---|
| ticket_id | 300 |
| passkey | 300Passkey |
| issuer_id | mitServiceBroker |
| sponsor_id | mitServiceBroker |
| redeemer_id | mitServiceBroker |

Table 2.2: Service Broker Ticket Values held by Scheduling Server

```
ticket_id : 200
passkey : 200Passkey
issuer_id : mitServiceBroker
CancelTicket(1001);
```

Figure 2-11: `CancelTicket()` Invocation

## 2.7 Ticket Cancellation

A Ticket can be cancelled by contacting the Service Broker that created it and passing the `ticket_id` of the Ticket to the `CancelTicket()` method. If the Lab Server that we described earlier wants to cancel a Ticket created by the Service Broker with `ticket_id = 1001`, it will invoke `CancelTicket()` as seen in Figure 2-11.

When `CancelTicket()` is invoked on a Service Broker, the Service Broker will examine the Header information and associate it with an iLab Entity. If the iLab Entity requesting that the Ticket be cancelled is not the Redeemer of the Ticket, the Service Broker will check to see if the Entity has the `"Cancel"` role for the Redeemer of the Ticket. If so, the Service Broker will invoke `CancelTicket()` on the Redeemer. If the Redeemer returns `true`, the Service Broker will consider the Ticket cancelled and it will return `true`. Otherwise the Service Broker will return `false`.

## 2.8 Creating, Giving Tickets

In Chapter 4, we will learn of a situation where a Scheduling Server might want to give a Ticket to a Lab Server. This Ticket would enable a Lab Server administrator to access the Scheduling Server and provide input into the scheduling process. In this situation, the Scheduling Server wants the Lab Server to have a Ticket, but does not

```
ticket_id : 300
passkey : "300Passkey"
issuer_id : mitServiceBroker
CreateAndGiveTicket(mitSchedulingServer, exp, pay, mitLabServer);
```

Figure 2-12: `CreateAndGiveTicket()` Invocation

want to grant the Lab Server Ticket creation permissions. This can be accomplished using the `CreateAndGiveTicket()` method.

The invocation of `CreateAndGiveTicket()` in Figure 2-12 indicates that the `mitSchedulingServer` is the Redeemer of the Ticket and the `mitLabServer` is the Entity that this Ticket should give given to. As a result, a Ticket will be created and then used as the only argument for the `InstallTicket()` method on the Lab Server. The values for `exp` and `pay` are not important for understanding the mechanics of how this method works.

## 2.9 General Ticketing, Kerberos

The Kerberos authentication protocol also supports credential based communication using the concept of a Ticket. However, Kerberos Tickets are not the same as General Tickets, and it is worthwhile to discuss the differences between them.

Kerberos Tickets enable users to authenticate themselves to kerberized service providers. Authentication is the first step in communicating with a kerberized service. Once a user's identity has been determined, the kerberized service will grant access to its services based on it. As such, a Kerberos ticket is used exclusively for user authentication.

General Tickets can be used for user authentication, but they can also be used in other ways. General Tickets represent resources and can be associated with individual identities, but this is not required. They describe the resources they represent with their payload. However, the content of a payload is not defined, and this allows a General Ticket to be used in a variety of ways. It is possible that the payload of a General Ticket will determine the authorization that should be granted to its

bearer. This, the main difference between a Kerberos Ticket is a General Ticket is that a General Ticket is extensible and has the ability to encapsulate authorization information.

# Chapter 3

# Scheduling Server

A Scheduling Server is an iLab Entity that provides experiment scheduling services to Lab Servers and Service Brokers. It delivers this functionality through a Web Application and Web Services. The Web Application allows lab owners, teachers, and students to interact with the scheduling process. Exactly how these users interact with the scheduling process is decided by the implementers of the Scheduling Server. For example, if a Scheduling Server supports an auction based system, it may have a Web Application that would allow a lab owner to create an auction and indicate a list of Service Brokers that will be allowed to participate in it. This same Web Application might allow a teacher on a participating Service Broker to supply details regarding the rules of this newly created auction. For example, the teacher might indicate that graduate students should receive twice as many bidding points as undergraduates. Finally, students could use the Web Application to place bids and view results.

Regardless of the scheduling algorithm employed, the goal of a Scheduling Server is to distribute available lab time to students. Under the General Ticketing mechanism, this is formally accomplished with the creation of a Lab Server Ticket. When the appropriate time arrives, the Scheduling Server supports a Web Service that allows Lab Server Tickets to be programmatically claimed by the students they were created for.

General Ticketing is used for all communication with the Scheduling Server. As we learned in Chapter 2, before this communication can take place, a series of Tickets

or ticketing permissions must be created. Section 3.1 will cover the details of how a Service Broker Domain must be configured before it can interact with a Scheduling Server.

The Scheduling Server grants access to its Web Application and Web Services through Tickets. As a consequence, implementers of a Scheduling Server must define a payload, or list of payloads, that they will support. These payloads are intentionally undefined, but there are certain rules they must adhere to. We will discuss these requirements and provide several examples in Section 3.2.

Scheduling Server Web Applications will vary greatly depending on the algorithms that they support. However, there are some basic requirements that all Web Applications must meet. These will be detailed in Section 3.3.

A Scheduling Server is a Service Provider in the iLab Framework, and it must support the Service Provider API. In addition to these standard methods, it also supports a method that allows students to claim Lab Server Tickets created on their behalf. The Web Services supported by the Scheduling Server will be presented in Section 3.4.

Once we have been introduced to the main pieces of the Scheduling Server, we will conclude by examining how a Scheduling Server will be used by a student to schedule and access lab time.

## 3.1 Domain Configuration

A Scheduling Server can provide its services to many different Service Broker Domains, and a single Service Broker Domain can contain many Scheduling Servers. For simplicity, we will consider how a Scheduling Server will interact with just a single Service Broker and Lab Server. Before it can be used, the following Tickets must be created and ticketing permissions granted:

- The Service Broker will create Service Broker Identification Tickets that it will use identify itself when it invokes Web Services on the Lab Server and Scheduling Server. Next, the Service Broker will create Service Broker Tickets for

the Lab Server and the Scheduling Server. The Service Broker will invoke `InstallTicket()` on the Lab Server and Scheduling Server to deliver each Service Broker Ticket. These two entities will use their Service Broker Ticket when they invoke Web Services on the Service Broker.

- The Scheduling Server must grant the `Create` and `Cancel` permissions to the Service Broker. This will allow the Service Broker to create Tickets for students and teachers, and then allow them to access the Scheduling Server's Web Application.

- The Lab Server must grant the `Create` and `Cancel` permissions to the Scheduling Server. This will allow the Scheduling Server to create Lab Server Tickets that grant time to a student.

- The Scheduling Server must have a Ticket created and delivered to a Lab Server. The Lab Server Administrator will use this Ticket to access the Scheduling Server's Web Application and provide input into the scheduling process.

These steps will have to be repeated for every Service Broker or Lab Server that will utilize the functionality of a Scheduling Server.

## 3.2 Scheduling Server Tickets

Tickets will be used to access the Web Application and Web Services of a Scheduling Server in the manner indicated by Section 2.2.1. The payload of these Tickets is not formally defined, but the chosen format must:

1. Provide a means for the Web Application to categorize the holder of the Ticket as either a lab owner, teacher, or student.

2. Provide a means for the Web Application to consistently identify a user between Web Application sessions.

```
<payload>
 <role>LabServerAdmin</role>
 <entity_id>mitLabServer</entity_id>
<payload>
```

Figure 3-1: Possible Lab Server Admin Payload for Scheduling Server Ticket

```
<payload>
 <role>Teacher</role>
 <entity_id>mitServiceBroker</entity_id>
 <group_id>10.467</group_id>
<payload>
```

Figure 3-2: Possible Teacher Payload for Scheduling Server Ticket

We have provided three concrete implementations of Scheduling Server payloads in Figures 3-1, 3-2, and 3-3. These implementations are encoded in the Extensible Markup Language, and differentiate between lab owners, teachers, and students using the `<role>` tag. They rely on the `<user_id>` tag to identify students between sessions. If a person requested access to the Web Application of a Scheduling Server, and presented a Ticket with the payload in Figure 3-1, the Scheduling Server would know to treat the holder as an administrator associated with `mitLabServer`. Similarly, if he or she presented a Ticket with the payload in Figure 3-3, the Scheduling Server would know to treat the holder as a student from the `mitServiceBroker` who has `group_id = 10.467` and `user_id = jsmith`. It is important to note that the payload of a Scheduling Server ticket is the only context from which a Scheduling Server can learn about its users.

```
<payload>
 <role>Student</role>
 <entity_id>mitServiceBroker</entity_id>
 <group_id>10.467</group_id>
 <display_name>John Smith</display_name>
 <user_id>jsmith</user_id>
<payload>
```

Figure 3-3: Possible Student Payload for Scheduling Server Ticket

```
void InstallTicket(Ticket t)
boolean CancelTicket(int ticket_id)
DateTime GetCurrentTime()
```

Figure 3-4: Service Provider API

## 3.3 Web Application Requirements

Due to the abstract nature of the Web Application, there are no strict requirements for it. The goal of the Web Application is to provide an interface into the scheduling process. However, this will manifest itself very differently if the Scheduling Server is supporting a auctioning system versus a first come, first served sign up process.

There are two general requirements of the Web Application. The first is that it must allow a Lab Server to indicate the times they wish to have scheduled. Lab Servers are free to utilize more than one Scheduling Server and, in order to avoid conflicts, Scheduling Servers must only distribute time that they have been told is available. If a Lab Server uses multiple Scheduling Servers, it must insure that the time intervals it offers to each are disjoint.

The second requirement is that the Web Application must provide a mechanism for a student to see the time that has been reserved for them. In theory, a Scheduling Server could be produced that would randomly schedule time among users without accepting any input from them. This Scheduling Server would be acceptable, so long as students could log into it and view the time that had been given to them. This is necessary because the Scheduling Server's Web Services do not provide any means to programmatically query for existing reservations.

## 3.4 Scheduling Server Web Services

The Web Service Application Programming Interface of the Scheduling Server supports the standard Service Provider methods. These are listed Figure 3-4 and described in Section 2.3.2.

The Scheduling Server supports one additional Web Service named `GetTicket()`.

```
http://schedserv/index.html?ticket_id=1&passkey=pw&issuer_id=mitSB
```

Figure 3-5: Scheduling Server Redirect

`Ticket GetTicket()`

This method can be invoked to claim a Lab Server Ticket for a user on a particular Lab Server. The Header of the SOAP Message in this Web Service invocation must include a `ticket_id`, `passkey`, and `issuer_id` of a Ticket whose payload includes information adequate to identify the student and the Lab Server he or she wants a Ticket for. When this method is invoked, the Scheduling Server will invoke `GetCurrentTime()` on the Lab Server and use this time to determine if a reservation is active. If so, a Lab Server Ticket will be returned. If not, `null` will be returned.

## 3.5 Standard Usage

Now that we have been introduced to the basic functionality of the Scheduling Server, we can take a closer look at how it will be used within a Service Broker Domain. The scenario below details the steps that a student will take in order to schedule lab time and then use a lab server.

1. The Student authenticates themselves at the Service Broker and uses its Web Application to indicate that he or she wants to schedule time on a particular Lab Server.

2. The Service Broker creates a Scheduling Server Ticket for the user and then redirects them to the Scheduling Server's Web Application with `ticket_id`, `passkey`, and `issuer_id` in the parameters of the URL. If the URL of the Scheduling Server was `http://schedserv/index.html` and the `ticket_id`, `passkey`, and `issuer_id` were equal to `"1"`, `"pw"`, and `"mitSB"`, respectively, then the redirect would appear as in Figure 3-5.

3. When the Scheduling Server's Web Application receives a request for usage, it parses the parameters and determines the associated `ticket_id`, `passkey`, and `issuer_id`. It invokes `VerifyTicket(1, pw)` on the Service Broker identified by `"mitSB"`, and learns the content of the Ticket. After parsing the payload, it grants the student access to the Web Application so he or she can schedule time. The Web Application may or may not offer immediate feedback to the student. If not, the student can repeat Steps 1 and 2 until he or she has finalized a reservation.

4. When the student's reservation time comes, he or she will log back into the Service Broker and request to launch the Lab Client. The Service Broker will create another Scheduling Server Ticket as it did in step 2, and invoke `GetTicket()` on the Scheduling Server with this newly created Ticket in the Header. The Scheduling Server will verify the contents of the Ticket, and use its payload to determine the student's identity and the lab server he or she wants a Ticket for.

5. The Scheduling Server will invoke `GetCurrentTime()` on the Lab Server and determine if the student has an active reservation. If so, it will return a Lab Server Ticket to the Service Broker.

6. If `GetTicket()` returns a Ticket, the Service Broker can give it to the Lab Client. If not, the Service Broker will know that the Student does not have access to the Lab Server presently.

7. The Lab Client can use this Ticket to access the Lab Server and run an experiment.

This scenario depicts how a student can reserve lab time, and then gain access to the Lab Server. With the steps clearly outlined, it is important to notice what each entity is responsible for. The Service Broker authenticates students and writes Tickets for them so they can use the Scheduling Server. The Scheduling Server accepts students redirected by the Service Broker and allows them to schedule time. The Lab Server waits until someone with a valid Lab Server Ticket wants to run an experiment.

There is clear separation of functionality between the entities, and each entity only delivers a single piece of functionality. By creating a Scheduling Server and using the General Ticket mechanism, we have introduced experiment scheduling without imposing a significant burden on the Service Broker or Lab Server.

# Chapter 4

# Polymer Crystallization Lab

We have re-implemented an existing online laboratory in order to ascertain the effectiveness of the designs we have proposed. The main goal of this work is to explore a specific Scheduling Server implementation that communicates using the General Ticketing mechanism. The online laboratory we chose to implement is known as the Polymer Crystallization Lab. It is interactive, and utilizes a first come, first served scheduling system. This chapter will begin by providing a general introduction to the lab. We will describe how the lab is presently utilized and enumerate its scheduling requirements. To support the lab, we needed to create a Service Broker and Lab Server, and we describe the most important aspects of each implementation. Finally, we will dedicate the majority of this chapter to describing our Scheduling Server implementation. We will indicate how it can be used, and how it operates within this new, interactive iLab infrastructure.

## 4.1 Introduction

The Polymer Crystallization Laboratory provides remote access to a polarized light microscope, a heating stage, and a digital camera. Students use these tools to heat a polymer sample above its melting point and then make observations as it subsequently cools. Specifically, these observations monitor crystal growth rates versus temperature. This "allows users to experimentally determine such properties as the

activation energy, the fold energy for the growing crystals, and the Avrami exponent of thin film crystallization."[5, p. 13]. This laboratory was implemented by Daniel Talavera, and is closely documented in his Master's Thesis [5]. His implementation is based on earlier work by Nasser [6] and Kuchling [7].

The lab was most recently used in the MIT course 10.467: *Polymer Science Laboratory* taught during the Fall of 2003. This course used the lab for a single, two-week long assignment. During this period, a total of 20 students were instructed to form groups of two to three people, and each group used the lab for approximately 4 hours. The 10.467 staff support the lab on a 24 hour per day basis, and the lab was consequently used for a total of 28 hours over a 338 hour (two week) period. This course is only taught in the Fall, and the lab goes unused for the remainder of the year.

### 4.1.1 Implementation Details

The Polymer Crystallization Laboratory (PCL) is delivered to students using a system of components. This system can be divided into three distinct parts: a Web Application, a Microscope Server, and a Graphical Lab Client.

**Web Application:** The laboratory uses a Web Application to authenticate users, reserve lab time, and provide access to the lab client. Student can interact with it using any standard web browser.

**Microscope Server:** The PCL provides a single software interface to its hardware using a component known as the Microscope Server. The server supports a text-based protocol, which can be used to convey both incoming commands and outgoing status updates. All communication with the server is sent over standard TCP/IP sockets using the Graphical Lab Client. The Microscope Server can only accommodate one user at a time.

**Graphical Lab Client:** Students use a graphical client to interact with the Microscope Server. This client is pictured in Figure 4-1. The job of the client is to accept input from the student in the form of pressing buttons and moving sliders and translate it into appropriate messages within the Microscope Server's

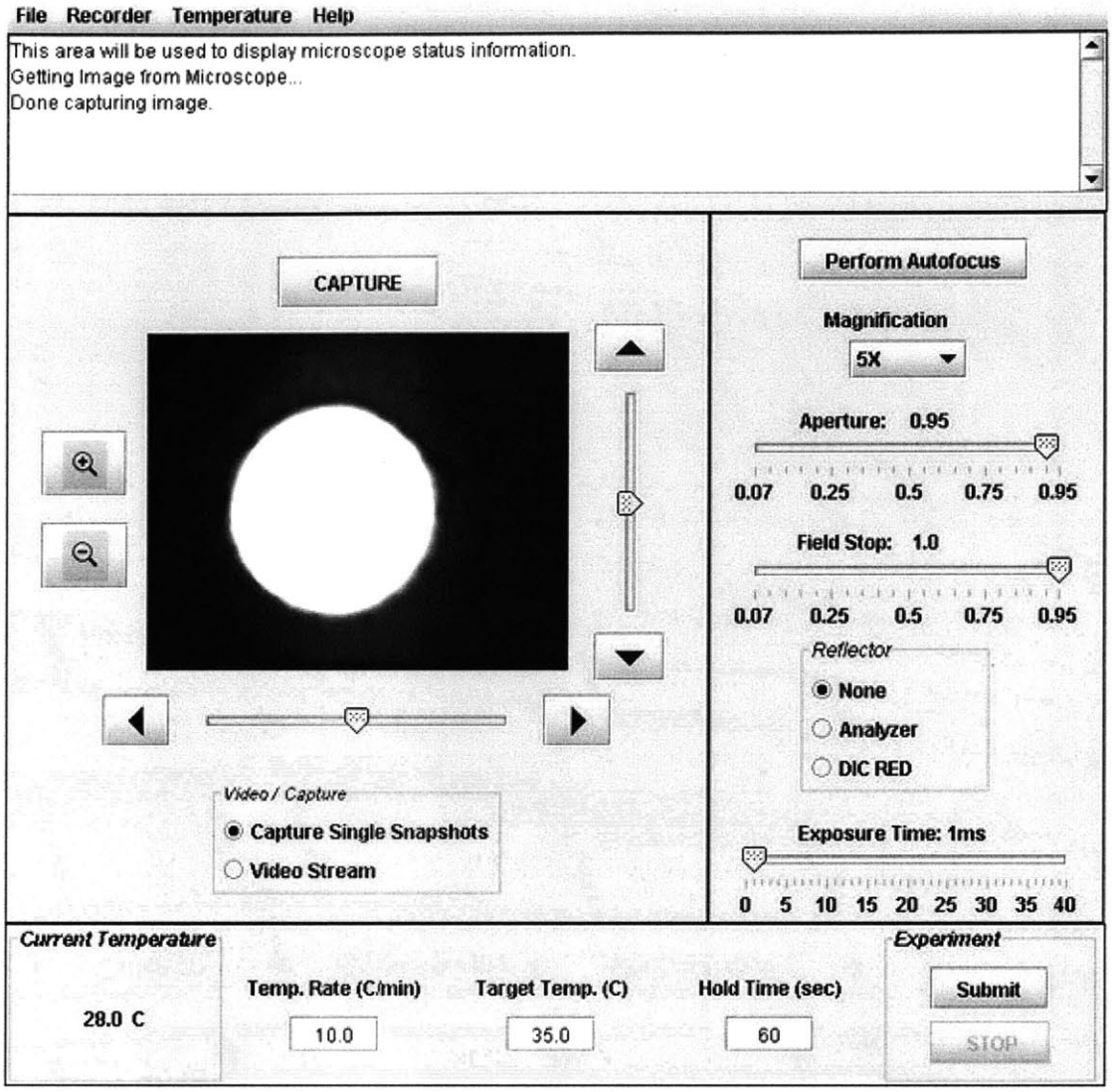


Figure 4-1: Polymer Crystallization Lab Client

protocol. The client also listens for status updates from the Microscope Server and updates itself to match the state of the Microscope Server.

### 4.1.2 Scheduling

The Polymer Crystallization Laboratory allows students to create advance lab reservations using a first come, first served scheduling system. Students interact with the scheduling system through the Web Application, and can create reservations to use

the lab 24 hours a day, 7 days a week. Any request they make is granted provided that it does not break either of the following rules:

- There can be only one active reservation at any time.
- The duration of a reservation must be equal to or less than 90 minutes.[1]

Finally, once a reservation has been created it cannot be cancelled. This summarizes the functionality offered by the PCL scheduling system. Even though it is fairly simple, it meets the needs of the PCL staff and their students. While our main goal is to create a Scheduling Server that can match this functionality, we are also interested in exploring mechanisms that can enhance this system. For example, this system can be abused: a pathological user could reserve an extended period of time by creating several consecutive reservations, as long as each individual reservation is less than 90 minutes. We will attempt to address this issue by building a simple language that administrators can describe rules with. Rather than providing administrators with a finite list of enforceable rules that they can turn on or off, we will give them ability to dynamically create rules as they see fit. Before we discuss the specifics of our Scheduling Server implementation, we must first understand the environment in which it operates.

## 4.2 Implementation Strategy

To implement this online experiment, we took four major steps:

- Created a Scheduling Server. This Scheduling Server supports first come, first served reservations. Its functionality exceeds that of the existing scheduling system.
- Created a Service Broker. This Service Broker will support the Service Broker API as documented in Section 2.3.1. In addition to writing and verifying Tick-

[1]The combination of a 90 minute maximum and 4 hours required to complete the assignment means that 10.467 students had to create multiple reservations to finish their work.

ets, the Service Broker is capable of: authenticating students, redirecting them to the Scheduling Server, and providing a Lab Client to the student.

- Modify the Remote Microscope Server. It will support the Service Provider API as documented in Section 2.3.2. Most importantly, the Microscope Server will be modified so that it can process Lab Server Tickets and grant lab access appropriately.

- Modify the Graphical Lab Client. It will be able to consume Web Services and use Tickets. It must be able to receive a Lab Server Ticket from the Service Broker and use it to communicate with the Lab Server.

The primary focus of this chapter is to discuss our implementation of a Scheduling Server. However, it is closely related to both the Service Broker and the Lab Server. Thus, we must first understand how these entities interact and communicate with one another. These interactions and communications can be specified by describing the Service Broker Domain that we implemented.

## 4.3 Service Broker Domain

We created a Service Broker Domain that was composed of a Service Broker, a Scheduling Server, and a Lab Server. The Entity IDs assigned to these servers were `mitServiceBroker`, `mitSchedulingServer`, and `mitLabServer`, respectively. Listed below are the configuration changes that we made after bringing our Service Broker Domain online:

- The Service Broker created Service Broker Identification Tickets and used these to identify itself when it invokes Web Services on the Lab Server and Scheduling Server. It also created and delivered a Service Broker Ticket to the Lab Server and the Scheduling Server. Each entity uses their Service Broker Ticket when they communicate with the Service Broker.

- The Scheduling Server was granted the Ticketing permissions `"Create"` and `"Cancel"` for the Lab Server. The Scheduling Server will use these permissions to create Lab Server Tickets when students make reservations, and cancel them if requested to. The Scheduling Server supports cancelling reservations even though the original PCL scheduling system does not.

- The Service Broker was granted the Ticketing permission `"Create"` for the Scheduling Server. The Service Broker will use this permission to create Scheduling Server Tickets that allow students and teachers to be redirected to the Scheduling Server's Web Application.

- A Scheduling Server Ticket was created and given to the Lab Server. This ticket will enable the Scheduling Server to access the Scheduling Server's Web Application.

### 4.3.1 General Communication

In the previous Batched Experiment architecture, a Service Broker allowed a student to launch a Lab Client. When the student ran experiments, the Lab Client would submit them to the Service Broker and the Service Broker would submit them to a Lab Server. In the new interactive experiment architecture this has changed. The new steps that a student will take to run an experiment are:

1. The student authenticates himself or herself at a Service Broker.

2. The Service Broker understands that the student needs to create a reservation on the Scheduling Server before he or she can access the Lab Server. At the student's request, the Service Broker creates a Scheduling Server Ticket and redirects him or her to the Scheduling Server's Web Application with the Ticket.

3. The student arrives at the Scheduling Server with a Ticket. The Scheduling Server verifies the Ticket with the Service Broker, and grants the student access. The student interacts with the Scheduling Server's Web Application and creates

a reservation. When a reservation is created, the Scheduling Server creates a Lab Server Ticket on the students behalf. The Scheduling Server will remember the student's reservation and the associated Lab Server Ticket.

4. When a student's reservation time arrives, the student returns to the Service Broker and indicates that they want to use the Lab Server.

5. The Service Broker has no knowledge of the student's reservation. However, it understands that the Scheduling Server will know if the student has a reservation. The Service Broker contacts the Scheduling Server and requests a Lab Server Ticket for the student.

6. The Scheduling Server remembers the student's reservation, and returns the Lab Server Ticket associated with it. If no such reservation exists, the Scheduling Server will not return a Ticket.

7. Upon receiving a Lab Server Ticket from the Scheduling Server, the Service Broker gives it to the student. Now the student can run an experiment because he or she can directly communicate with the Lab Server.

This is a high level view of how a student reserves time and then uses a Lab Server to run an experiment. Once we have been introduced to the details of our implementation, Section 4.5 will retrace these steps again in greater detail.

### 4.3.2 Service Broker Implementation

The Service Broker we implemented supports the Service Broker APIs discussed in Section 2.3.1. All of the methods that were previously used as "pass through" methods have been removed because they are no longer necessary in an interactive experiment architecture.

In the previous version of the iLab infrastructure, a Service Broker acknowledged two separate roles: users and administrators. Our implementation of the Service Broker needs to recognize a new role in addition to these two: teachers. Teachers will be

associated with groups just as students are and they will not be granted any administrative abilities on the Service Broker. However, the Service Broker must provide a means by which a teacher can visit a Scheduling Server. When the teacher exercises this ability, the Service Broker creates a Scheduling Server Ticket that identifies him or her as a teacher. When he or she arrives on the Scheduling Server, he or she will be empowered to define rules that will affect the sign up process for all of the students in their group.

### 4.3.3 Lab Server Implementation

The Lab Server we implemented supports the Service Provider APIs discussed in Section 2.3.2. All other Web Service methods were removed from the API as they are not necessary for an interactive experiment.

It was determined that Web Service based communication would not be able to support the high-bandwidth requirements of the Polymer Crystallization Lab. As a consequence, it was decided that Web Services should be used at the beginning of an experiment to serve as a "handshake," but further communication will rely on a TCP/IP socket based connection. This combination allowed us to rely on the authentication scheme of General Ticketing, while taking advantage of the faster, legacy communication mechanism. A new Web Service was introduced to the Lab Server API named `Connect()`.

`string Connect()`

This method will be invoked by a Lab Client. The Lab Client will include information identifying a Lab Server Ticket in the Header of the method invocation. A Lab Server will be able to verify the content of this Ticket with the Service Broker, and use the results to determine if a user should be granted access to the lab. If so, the Lab Server will return a `string` to be used as a password. The Lab Client will then open up a socket connection to a well-known port on the Lab Server and authenticate itself by providing the `string` result.

## 4.4 Scheduling Server Implementation

Our Scheduling Server can be used by Lab Server administrators, teachers, and students. Lab Server administrators can use a Scheduling Server to describe the times that they want to make available. For example, a Lab Server administrator could say, "My lab will be available Monday through Friday, between 9AM and 5PM starting on February 3rd and ending on May 13th." They can also tell the Scheduling Server who should be able to receive this time as in, "Any student from MIT can use the lab, and students from CalTech that are in the course CHEM147: *Polymer Chemistry* should also be allowed to use the lab." We will refer to this type of rule as a *Recipient Rule.* Finally, a Lab Server administrator could tell the Scheduling Server rules that it wants every reservation to conform to, such as "At a minimum, people must sign up for 30 minutes of time, but they should never exceed 90 minutes." We will refer to this type of rule as a *Sign Up Rule.* Once a Lab Server administrator has specified this information, the Scheduling Server can now be used by teachers and students. A Scheduling Server refers to the combination of a time period, Recipient Rules, and Sign Up Rules as a *Time Distribution.* Time Distributions are created by, and ultimately belong to, Lab Server administrators.

To use a Scheduling Server a teacher or student must be redirected to it and supply a valid Scheduling Server Ticket. Figures 4-3 and 4-4 contain Scheduling Server payloads we used in our implementation to identify teachers and students. By parsing these payloads, a Scheduling Server can associate information such as the Service Broker or Group with each user. This information can be used to determine if the user meets the criteria specified by the Recipient Rules of a Time Distribution. If a user does meet this criteria, we say that the user *qualifies* for the Time Distribution.

In our example, a teacher redirected to the Scheduling Server would have to be from MIT or from CalTech and associated with CHEM147 to qualify for the Time Distribution. Once a teacher qualifies for a Time Distribution, they can view all of the information pertaining to it, including the Recipient and Sign Up Rules. A teacher can append additional Recipient and Sign Up Rules to the Time Distribution.

```
<payload>
 <role>LabServerAdmin</role>
 <entity_id>mitLabServer</entity_id>
<payload>
```

Figure 4-2: Scheduling Server Ticket: Lab Server Payload

```
<payload>
 <role>Teacher</role>
 <entity_id>mitServiceBroker</entity_id>
 <group_id>10.467</group_id>
<payload>
```

Figure 4-3: Scheduling Server Ticket: Teacher Payload

For example, the teacher could say, "I know my assignment is very short and none of my students will need more than 60 minutes in the lab." Any rules supplied by a teacher will only be applied to students that are from their Service Broker and in their Group.

In order for a student to qualify for a Time Distribution, they must meet the criteria specified by the Recipient Rules supplied by a Lab Server administrator and those specified by a teacher from their group. If they qualify, they will be allowed to sign up for time.

To gain a better understanding of how our Scheduling Server implementation works, we will formalize the concepts of Time Distributions, Recipient and Sign Up Rules.

```
<payload>
 <role>Student</role>
 <entity_id>mitServiceBroker</entity_id>
 <group_id>10.467</group_id>
 <display_name>John Smith</display_name>
 <user_id>jsmith</user_id>
<payload>
```

Figure 4-4: Scheduling Server Ticket: Student Payload

### 4.4.1 Time Distributions

The Scheduling Server allows Lab Server administrators to create Time Distributions by specifying a period of time, a list of Recipient Rules, and a list of Sign Up Rules. The rules specified by a Lab Server administrator affect everyone who interacts with a Time Distribution. In particular, the Recipient Rules determine which teachers can interact with it. Each teacher that qualifies to interact with the Time Distribution can attach Recipient and Sign Up Rules of their own to it. The rules specified by a teacher only affect students that are from the same Service Broker and share the same group as the teacher. If a Lab Server administrator fails to specify any rules, then anyone can receive time and create reservations.

The time period described in a Time Distribution is composed of a start date, and end date, and a weekly recurrence pattern. In the example above, the start and end dates would be February 3rd, 2004 and May 13th, 2004. The recurrence pattern would be 9AM to 5PM on Monday through Friday with no time available on Saturday and Sunday. February 3rd, 2004 is a Tuesday, so the first time available in this Time Distribution would be at 9AM. May 13th, 2004 is a Thursday, so the last reservation would have to end at 5PM.

A Scheduling Server will allow a Lab Server to create multiple Time Distributions, so long as the available times allowed by the individual Time Distributions do not overlap. For example, the Lab Server administrator might create a Time Distribution that was active only on the weekends between February 3rd and May 13.

### 4.4.2 Rules

When a teacher or student begins a new session with the Scheduling Server's Web Application, their Ticket payload is parsed and inserted into a map data structure. For example, if a student arrived at the Scheduling Server with a Ticket containing the payload in Figure 4-4, then the map in Figure 4-5 would be created. This map links general attributes of every user to specific values of a particular user. It is known as an Attribute Value map.

```
{
"USER_ID" -> "jsmith",
"GROUP_ID" -> "10.467",
"DISPLAY_NAME" -> "John Smith",
"SERVICE_BROKER_ID" -> "mitServiceBroker"
}
```

Figure 4-5: Student Attribute Value map

Recipient Rules and Sign Up Rules are specified using a simple text format. An example of a rule is `(SERVICE_BROKER_ID EQUAL_TO "mitServiceBroker")`. All rules follow the general format `(Attribute Predicate Value)`, and evaluate to either `true` or `false`. Rules must be evaluated within the context of a user's Attribute Value map. To evaluate a rule above within the context of Figure 4-5, we query the map for the key associated with `SERVICE_BROKER_ID` and receive `"mitServiceBroker"`. We then apply the `EQUAL_TO` predicate to both `strings`, and the rule evaluates as `true`. The Scheduling Server supports eight predicates that can be used in rules, and they are listed in Table 4.1. All of the predicates operate on `strings` and all of them return either `true` or `false`. The predicates `BEFORE` and `AFTER` can be used with input that represents a date, but it must match the `"MM/DD/YYYY"` format.

It also supports six attributes listed in Table 4.2. When rules are specified, they are either Recipient Rules or Sign Up Rules. Recipient Rules are enforced when a user first arrives at the Scheduling Server. A Recipient Rule can use the all of the attributes except `DURATION`. In the Recipient Rule context, the `DATE` attribute is assigned the current date. Sign Up Rules are enforced when a student request to create a reservation. They can contain all of the attributes. The `DURATION` attribute represents the number of minutes the student has requested to reserve. In the Sign Up Rule context, the `DATE` attribute is assigned the date of the requested reservation. Regardless of context, `DATE` will always be converted in a properly formatted `string` so that it can be used with `BEFORE` or `AFTER`.

Finally, If $\psi$ and $\phi$ are rules, then ( $\psi$ `AND` $\phi$ ) and ( $\psi$ `OR` $\phi$ ) are rules as well. ( $\psi$ `AND` $\phi$ ) evaluates to `true` if and only if both $\psi$ and $\phi$ evaluate to `true`,

| Name | Function | Usage Example |
|---|---|---|
| EQUAL_TO | String Equality | (GROUP_ID EQUAL_TO "10.467") |
| NOT_EQUAL_TO | String Inequality | (GROUP_ID NOT_EQUAL_TO "10.467") |
| LESS_THAN | Numeric or String Comparison | (DURATION LESS_THAN "90") |
| LESS_THAN_OR_EQUAL_TO | Same | Similar to above |
| GREATER_THAN | Same | Similar to above |
| GREATER_THAN_OR_EQUAL_TO | Same | Similar to above |
| BEFORE | Date Comparison | (DATE BEFORE "5/17/2004") |
| AFTER | Date Comparison | (DATE BEFORE "5/17/2004") |

Table 4.1: Valid Predicates

| Attribute | Value |
|---|---|
| SERVICE_BROKER_ID | User's Service Broker ID |
| USER_ID | User's User ID |
| GROUP_ID | User's Group ID |
| DISPLAY_NAME | User's Display Name |
| DATE | Context Dependent |
| DURATION | Duration of Requested Reservation |

Table 4.2: Valid Attributes

( $\psi$ OR $\phi$ ) evaluate to true if either $\psi$ and $\phi$ evaluate to true.

Now that we have a deeper understanding of Time Distributions, Recipient Rules, and Sign Up Rules, we will examine how the Scheduling Server allows this information to be entered.

### 4.4.3 Web Application Interface

Whenever anyone arrives at our Scheduling Server, he or she is presented with a list of Time Distributions. Students and teachers are shown all of the Time Distributions for which they qualify. It is possible that he or she will not qualify for any Time Distributions, in which case they cannot interact with the Scheduling Server. Lab Server administrators are presented with a list of Time Distributions that they have

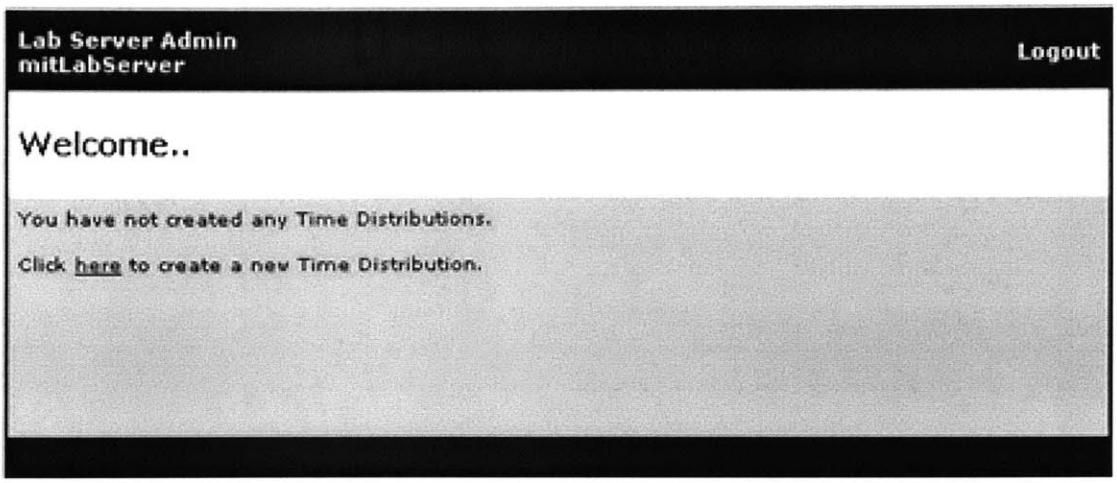


Figure 4-6: Lab Server Administrator: Main Page, No Time Distributions

created. If they have not created any Time Distributions, they will be allowed to create a new one (See Figure 4-6).

If a Lab Administrator follows the link in Figure 4-6, he or she will be redirected to a page where they can enter information necessary to create a Time Distribution. This includes specifying a time period, Recipient Rules, and Sign Up Rules. Figure 4-7 displays a page that has been completed by a Lab Server administrator so that it represents Time Distribution we described at the beginning of this chapter.

This Lab Server administrator has associated both Recipient and Sign Up Rules with this Time Distribution. These Rules are displayed in Figures 4-8 and 4-9 to allow for closer inspection. The Scheduling Server provides some limited rule input checking. It will not allow you to enter ill-formed rules, use invalid predicates, or use invalid attributes.

As a side note, as soon as the Lab Server administrator saves this information, students from MIT and CalTech in the CHEM147 will be able to sign up for time. Additionally, teachers from MIT and the CalTech CHEM147 will be able to append their own rules to the Time Distribution.

After saving the Time Distribution, the Lab Server administrator is returned to the page they began on. This page previously indicated that no Time Distributions had been created by the Lab Server administrator, but now it has been updated to

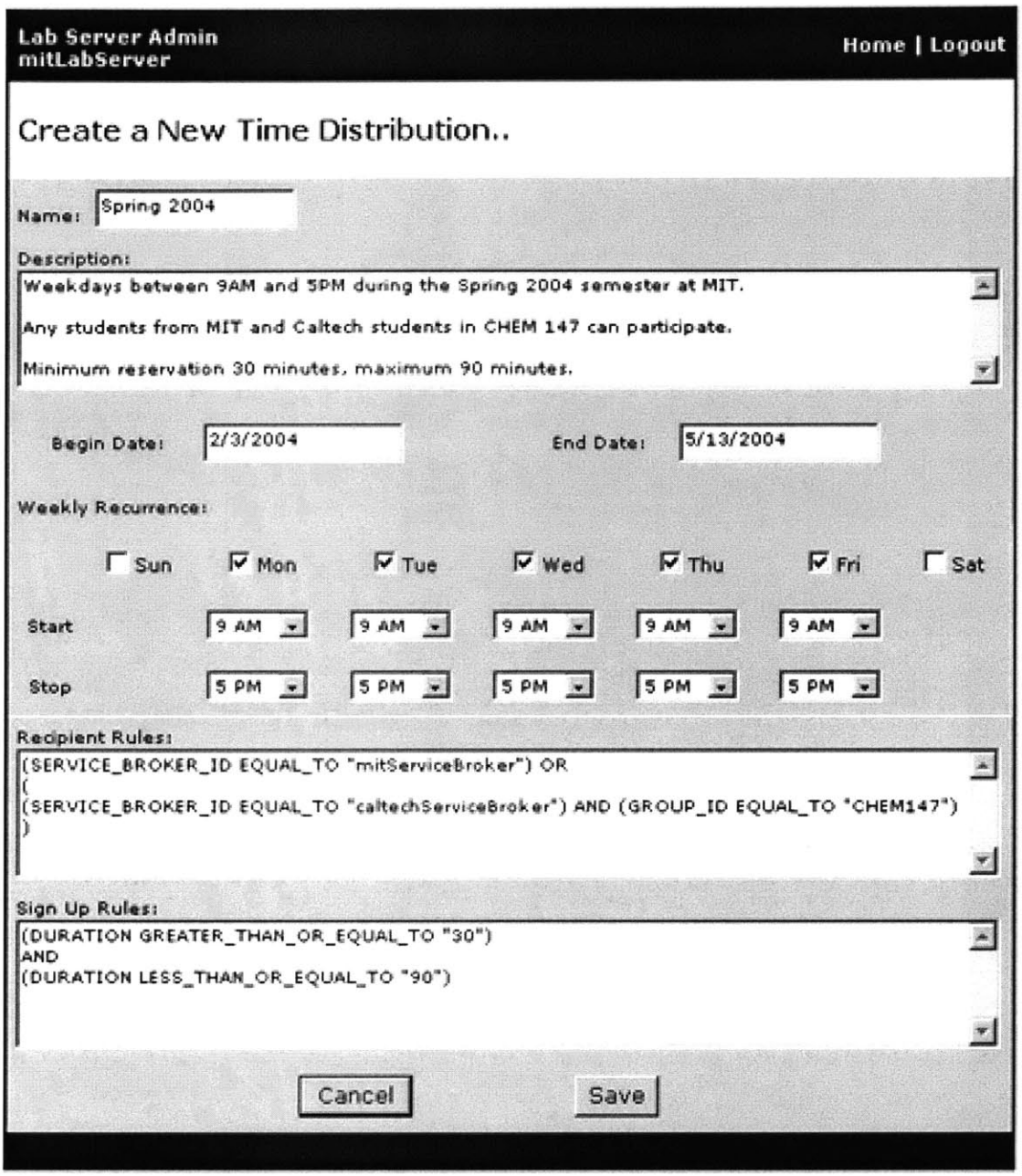


Figure 4-7: Lab Server Administrator: Creating a Time Distribution

```
((SERVICE_BROKER_ID EQUAL_TO "mitServiceBroker")
 OR
   ((SERVICE_BROKER_ID EQUAL_TO "caltechServiceBroker")
   AND
   (GROUP_ID EQUAL_TO "CHEM147")))
```

Figure 4-8: Lab Server Recipient Rules

```
((DURATION GREATER_THAN_OR_EQUAL_TO "30")
  AND
  (DURATION LESS_THAN_OR_EQUAL_TO "90"))
```

Figure 4-9: Lab Server Sign Up Rules

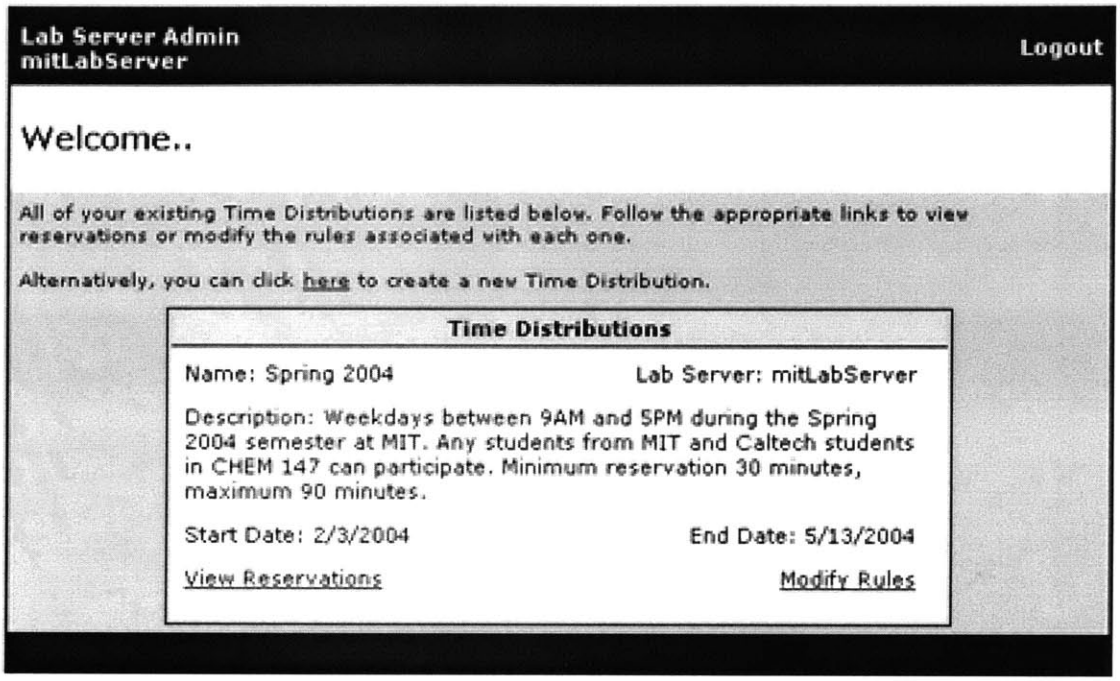


Figure 4-10: Lab Server Administrator: Main Page, One Time Distribution

summarize the newly created Time Distribution (see Figure 4-10).

If the Lab Server administrator follows the "Modify Rules" link, they will be able to update the Recipient and Sign Up Rules associated with the distribution. If they select the "View Reservations" link, they will be allowed to view the reservations that have been made that day. Reservations are displayed in a tabular format, and a Lab Server administrator can see all of the information associated with every reservation (see Figure 4-11).

Lab Server administrators, teachers, and students all view reservations in the format presented in Figure 4-11. However, a teacher will only be able to see the details associated with students from their group; if a teacher is associated with the group 10.467 and he or she views the page in Figure 4-11, they would only see the word "Reserved" during Jane Doe's reservation times. Students are only allowed to

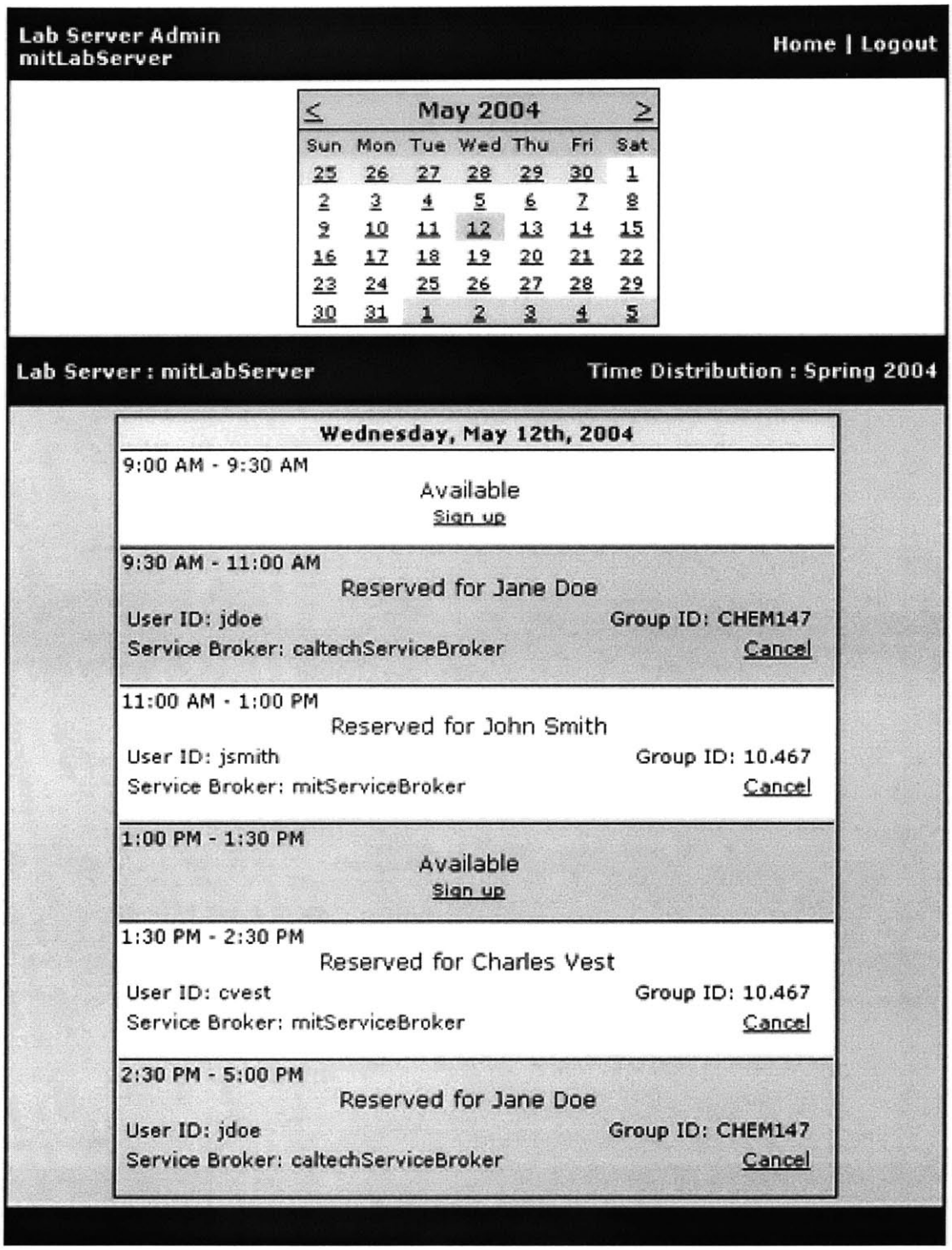


Figure 4-11: Lab Server Administrator: Viewing Reservations

see reservation details for reservations they have previously made. Anyone who can view a reservation can cancel it. In effect, a Lab Server administrator can cancel any reservation in one of their Time Distributions, a teacher can cancel any of their student's reservations, and students can only cancel their own reservations.

When a teacher arrives at the Scheduling Server, he or she will be presented with a list of Time Distributions that he or she meets the Recipient Rules for. This will look very similar to 4-10. If they select "Modify Rules", for a particular Time Distribution, he or she will be taken to a page that describes the Time Distribution and allows them to add or modify Recipient and Sign Up Rules (see Figure 4-12). This figure shows a teacher that has created an additional Sign Up Rule. This rule makes it impossible for her students to create reservations that are longer than 60 minutes.

When a student arrives at the Scheduling Server, they will see a list of all the Time Distributions that they qualify for, just as the teacher did. However, they will not have the option to "Modify Rules" for any distribution, instead "View Reservations" from Figure 4-10 will be replaced with "Create Reservations." Clicking on this link will lead the student to the view the same reservation view page that is in Figure 4-11. The student can create a new reservation by select the "Sign up" link on any time period that is available. When they select this link, the link text changes to a plain label that reads "Enter Duration," a drop down list appears to the right of the label, and a "Create Reservation" link appears to the right of the combo box. The student can select from the drop down box to indicate the duration of their reservation. They can select the link to create the reservation (see Figure 4-13).

## 4.5 Creating a Reservation, Using a Lab Server

The previous section explained how Lab administrators, teachers, and students interact with the Web Application. Now that we have an understanding of how a Time Distribution can be specified and are more familiar with what the Web Application looks like, we will follow the steps that a steps that a student will take to create a

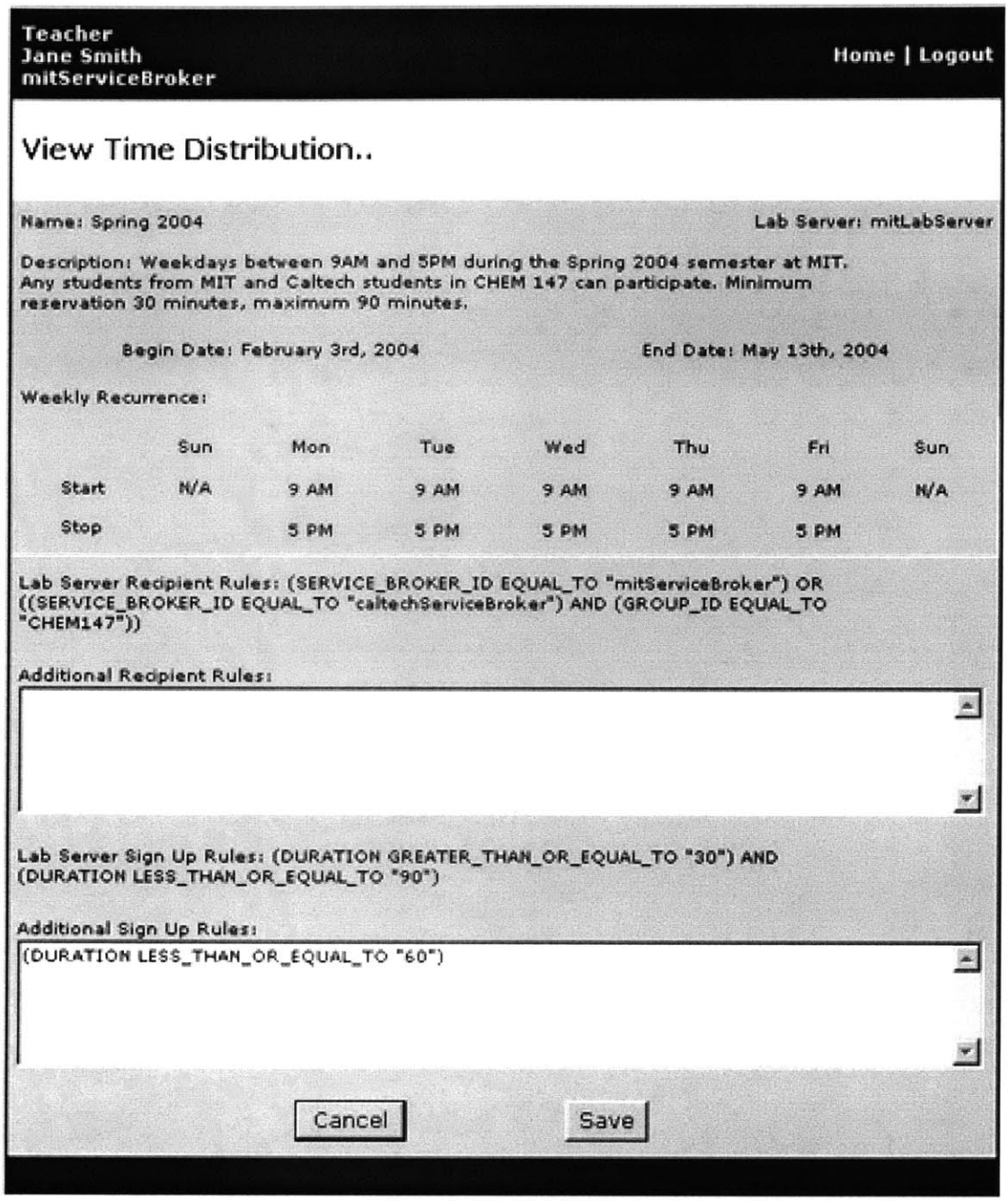


Figure 4-12: Teacher: Modifying Distribution Rules


Figure 4-13: Student: Creating a Reservation

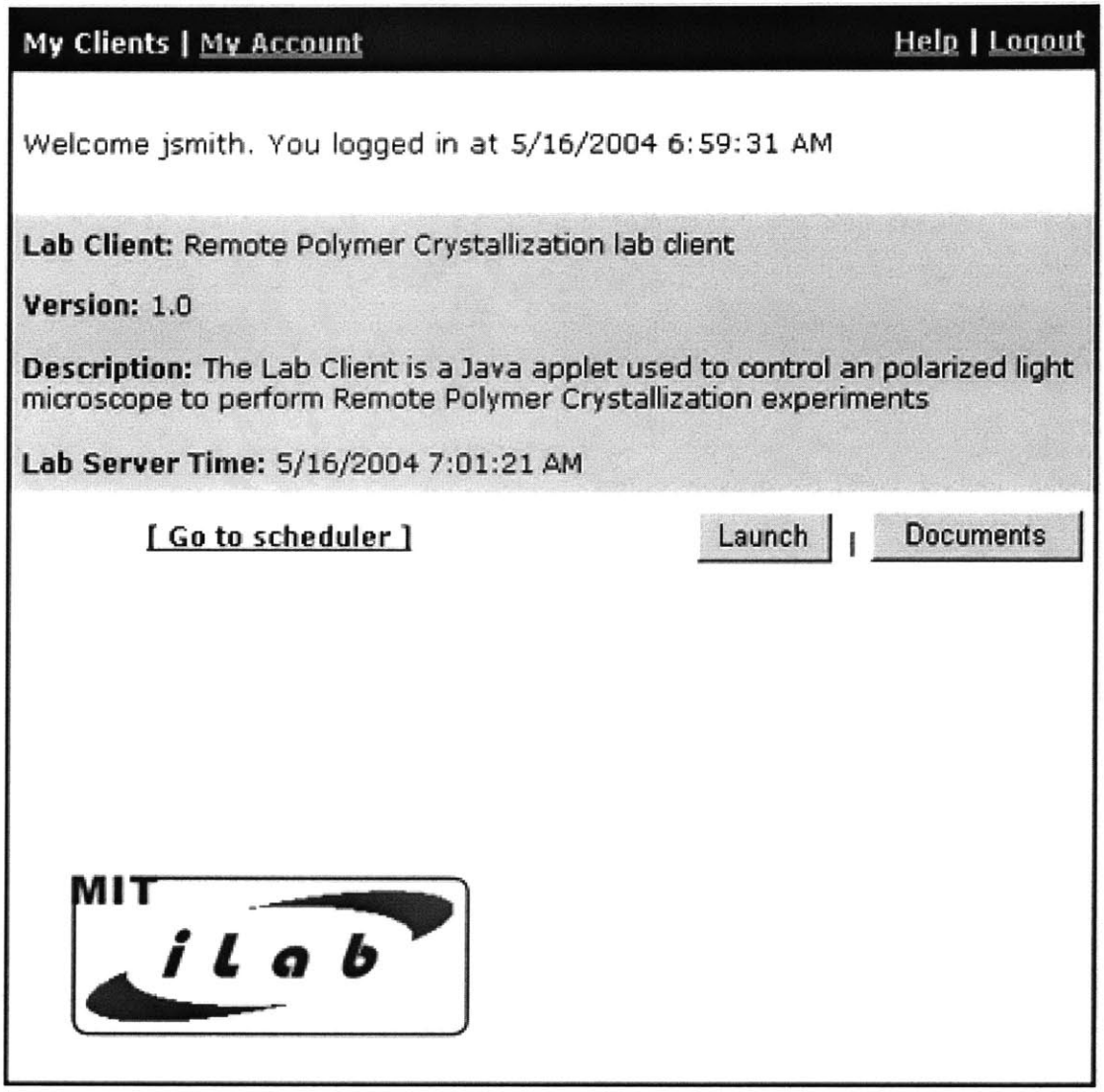


Figure 4-14: Student: Service Broker Main Page

reservation for lab time and redeem it on the Lab Server using our implementation. These steps mirror those that were presented in Section 4.3.1, but they are described in more detail below.

1. The student authenticates themselves at the Service Broker. After successfully logging in, he or she is presented with the page in Figure 4-14. The student belongs to the group 10.467 that is on the `mitServiceBroker`.

2. The student knows that he or she needs to schedule time before they can launch

```
https://ilab/m.html?ticket_id=10&passkey=pw&issuer_id=mitServiceBroker
```

Figure 4-15: Scheduling Server URL Redirect

the Client, so he or she clicks the "Go to scheduler" link.

3. The Service Broker creates a Scheduling Server Ticket for the student. The payload exactly matches Figure 4-4. The Service Broker redirects the student to the Scheduling Server's URL, supplying the `ticket_id`, `passkey`, and `issuer_id` as parameters of the URL. The resulting URL will be similar to Figure 4-15.

4. The Scheduling Server receives a request to display a page, and it parses the parameters of the URL to determine information that can identify the embedded Ticket. The Scheduling Server does not know what the payload of the Ticket contains, so it invokes `VerifyTicket(10,"pw")` on the Service Broker.

5. The Service Broker responds to the `VerifyTicket()` invocation by returning the Ticket it created for the student.

6. Now the Scheduling Server understands the meaning behind the Ticket. It creates an Attribute Value map for the student, exactly as seen in Figure 4-5.

7. The Scheduling Server then checks to see if the student qualifies for any Time Distributions. The student qualifies for one, so they are shown a screen similar to Figure 4-10. The only difference is that information identifying the student appears the upper left hand corner and the links "Click here to create a new Time Distribution" and "Modify Rules" are not visible. The "View Reservations" link reads "Create Reservation," and the student selects it.

8. The Scheduling Server redirects the user to a page that looks like Figure 4-11. The student can create a new reservation by select the "Sign up" link on any time period that is available. When they select this link, the link text changes to "Enter Duration," a drop down list appears to the right of the link, and a "Create Reservation" link appears to the right of the combo box. The drop

down list allows the user to indicate the duration of the reservation in 30 minute intervals.

9. The student selects 60 minutes and selects the "Create Reservation" link.

10. The Scheduling Server recognizes that a 60 minute reservation is being requested on the Time Distribution. It adds `DURATION -> "60"` to the student's Attribute Value map and begins to evaluate the Sign Up rules. The Sign Up Rules associated with the Time Distribution by the Lab Server administrator are evaluated first. These rules are listed in Figure 4-9, and they evaluate to `true` because the student's requested reservation is greater than or equal to 30 minutes and less than or equal to 90 minutes. Next, the Scheduling Server evaluates the Sign Up Rules that were associated with the Time Distribution by the teacher who belongs to Group 10.467 in Figure 4-12. These rules also evaluate to `true` because the student's requested reservation is less than or equal to 60 minutes.

11. The student has satisfied the Sign Up Rules associated by both the Lab Server administrator and his teacher. The Scheduling Server will create the reservation. The Scheduling Server contacts the Service Broker and invokes `CreateTicket( "mitLabServer", exp, pay)` where `exp` is an expiration date, and `pay` is a Lab Server Ticket payload with the reservation's start and end times, and the student's user id and group. The Scheduling Server records the student's reservation information, along with all of the information pertaining to the Lab Server Ticket.

12. The student waits for the time of their reservation. Once it arrives, he or she revisits the Service Broker and sees the page in Figure 4-14. They select the "Launch" button.

13. The Service Broker has no knowledge of the student's reservation. However, it understands that the Scheduling Server will know if the student has a reservation. The Service Broker creates another Scheduling Server Ticket that contains

a payload as seen in Figure 4-4. Then the Service Broker invokes `GetTicket()` on the Scheduling Server, inserting the newly created `ticket_id`, `passkey`, and `issuer_id` into the Header of the invocation.

14. The Scheduling Server extracts the Ticket identifying information out of the Header of the `GetTicket()` invocation. The Scheduling Server does not know what the payload of the Ticket contains, so it invokes `VerifyTicket()` on the Service Broker.

15. The Service Broker responds to the `VerifyTicket()` invocation by returning the Ticket it created for the student. This gives the payload seen in Figure 4-4 to the Scheduling Server.

16. Now the Scheduling Server understands the meaning of the Ticket. It checks to see if it has previously recorded a reservation for the user, and it has. It returns the Lab Server Ticket that it created when the reservation was made.

17. Upon receiving a Lab Server Ticket from the Scheduling Server, the Service Broker allows the student to launch a Graphical Lab Client. It does by creating a new browser window and writing a custom `<applet>` in it. The tag it writes contains the `ticket_id`, `passkey`, and `issuer_id` of the Lab Server Ticket (see Figure 4-16).

18. The Graphical Lab Client can parse this parameter information, and use it to invoke `Connect()` on the Lab Server.

19. The Lab Server extracts the Ticket identifying information out of the Header of the `Connect()` invocation. The Lab Server does not know what the payload of the Ticket contains, so it invokes `VerifyTicket(36,"p$y4o5x_k6jq8{w7bm")` on the Service Broker.[2].

20. The Service Broker responds to the `VerifyTicket()` invocation by returning the Lab Server Ticket. The payload of this Ticket indicates that start and end

[2]"p$y4o5x_k6jq8{w7bm" is a Ticket passkey directly from our actual implementation

```
<applet>
  <param name="archive" value = "signedapplet.jar"/>
  <param name="code" value = "client/scopeformapplet.class "/>
  ...
  <param name="ticket_id" value="36"/>
  <param name="passkey" value="p$y4o5x_k6jq8{w7bm/"/>
  <param name="issuer_id" value="mitServiceBroker"/>
</applet>
```

Figure 4-16: `<applet>` Tag

times that the user can access the Lab Server for.

21. Now the Lab Server understands the meaning of the Ticket. It checks to see that its current time is between the start and end time of the ticket. Seeing that it is, it allows the student to run the experiment by returning a password. The Lab Client uses this password to connect to the TCP/IP based socket of the Lab Server.

# Chapter 5

# Conclusion

We learned a number of important lessons from implementing our design. Many of these resulted in expected strengths, but we also discovered areas that need additional work.

## 5.1 General Ticketing Strength

The greatest improvement of our design was the General Ticketing mechanism. The extensible nature of Ticket payloads allows Tickets to be used in many different ways. For example, Lab Server Tickets form the basis of scheduling by enabling experiment time to be commoditized and distributed. Simultaneously, Scheduling Server payloads allow a Service Broker to collect user information and transfer it to a remote Web Application. The Scheduling Server is a fully functioning Web Application without a login screen. This is achieved using Tickets. Once aware of the user's information, a Scheduling Server can process student requests and enforce specified rules. Finally, the most fundamental Tickets we implemented were those associated with a Service Broker Ticket and Service Broker Identification Tickets. These Tickets were used simply to authenticate the identity of a Web Service invoker. From a design perspective, we believed that General Ticketing would be powerful due to its flexibility; however we were surprised at how the flexibility enabled us, as implementers, to quickly develop meaningful payloads through for iLab Entities.

```
<payload>
 <role>Teacher</role>
 <entity_id>mitServiceBroker</entity_id>
 <group_id>10.467</group_id>
<payload>
```

Figure 5-1: Teacher Payload Example

```
https://sched/index.html?ticket_id=1&passkey=test&issuer_id=mitSB
```

Figure 5-2: Scheduling Server URL Redirect

## 5.2 General Ticketing Weakness

However, the General Ticketing mechanism is not perfect. During our implementation, it became clear that the mechanism could be made more efficient. Consider the situation where the Service Broker has written a Scheduling Server Ticket for a teacher and is redirecting them to the Scheduling Server. The payload of the Scheduling Server Ticket can be seen in Figure 5-1. The URL redirect can be seen in Figure 5-2.

When the Scheduling Server receives the request to view the URL, it will parse the parameters and invoke `VerifyTicket(1, test)` on the Service Broker. The Service Broker will respond by returning a Ticket that has the payload described in Figure 5-1. Now that the Scheduling Server has the contents of the Ticket, it recognizes that the user is a Teacher associated with 10.467. If the Service Broker is trusted by the Scheduling Server, it should be able to deliver this information directly to it. For example, it should be able to redirect the teacher to the Scheduling Server as seen in Figure 5-3. In this URL redirect, the Service Broker contacts that the user and directly indicates that a Teacher who belongs to group 10.467 is being redirected..

Based on our implementation, this extra round trip did not introduce any noticeable latency, but all of our iLab Entities were located very close to one another (in

```
https://sched/index.html?role=Teacher&group=10.467
```

Figure 5-3: Scheduling Server URL Redirect

the network sense). This particular issue should be considered closely before General Ticketing is accepted as an official part of the iLab infrastructure. Further, we wonder whether or not investigating public key encryption could benefit our Ticket verification system. Tickets could be modified to contain an additional `signature` field, which would contain the Service Broker's private key signature of the information contained in the Ticket. Newly received Tickets could be verified without a network round trip.

## 5.3 Scheduling Server Weakness

After designing and implementing the Scheduling Server, we feel its design and requirements could benefit from further definition. The design defines only two general requirements, and both are open to interpretation. This results in wide latitude for implementers of a Scheduling Server, but it also fails to provide any helpful implementation guidelines. One of the general requirements for the Scheduling Server from Chapter 3 is that "it must allow a Lab Server to indicate the times they wish to have scheduled." When we implemented this aspect of the Scheduling Server, we decided to describe time periods using a start date, and end date, and a weekly recurrence pattern. This decision was made in an arbitrary fashion: the results suited our purposes, it was easy to describe time intervals, and it was simple to implement. Because of the requirement that a Lab Server must be able to indicate their available time, anyone who implements a Scheduling Server will encounter this problem.

When we initially approached the problem of describing a time period, we looked at the functionality provided by the *Internet Calendaring and Scheduling Core Object Specification*[11]. This specification was first proposed in 1998 by the Internet Engineering Task Force, and has since become an industry standard. It is used in popular programs such as Apple's iCal and Microsoft's Outlook. The specification covers more than how to describe a period of time, but a portion of the standard is explicitly devoted to this problem [11, Sec 4.3.9].

Standardizing time descriptions may provide more benefit than delivering a spe-

cific guideline to a Scheduling Server implementers. If all Scheduling Servers use the same descriptions of time, then we will have a means for programmatically discovering what Lab Server times are available from Scheduling Servers. To date, the iLab infrastructure has not considered scalable solutions towards the goal of automating lab discovery. Research in this area may provide some interesting insight.

## 5.4 Future Work

The design of our Scheduling Server implementation was based on the requirements of three existing online laboratories. The Residence Exchanger, the Heat Exchanger administered by Siddharta Sen, and the Polymer Crystallization Lab administered by Derik Pridmore.[1] Each of these online experiments required a scheduling system and when we examined the systems they had created, we found three distinct implementations of an HTML based, first come, first served sign up system. Further, all of these systems offered very basic functionality and it became clear that lab implementers want to focus their efforts on enabling online labs and solving domain dependent issues in that vein. They are not interested in building scheduling systems.

In terms of immediate work, we will benefit ourselves and our community by creating a well-tested, professionally implemented version of a first come, first served Scheduling Server. This Scheduling Server become a freely available reference implementation that is part of the iLab suite. If we consider the functionality demonstrated by the Scheduling Server prototype of Chapter 4 as a base line, then we have a number of suggestions in mind for implementation specific extensions:

### 5.4.1 Dynamic Attributes

We implemented a number of attributes that could be used in both Recipient and Sign Up Rules. In our opinion, the most interesting attribute we put forth was `DATE` because its value was generated dynamically. The Scheduling Server knows

[1]For more information on these labs, see http://www.vs-c.de/, http://heatex.mit.edu, and http://polymerlab.mit.edu, respectively

```
(((LAST_NAME LESS_THAN_OR_EQUAL "N") AND (DATE BEFORE "2/14/2004"))
 OR
 ((LAST_NAME GREATER_THAN "N")
   AND (DATE AFTER "2/14/2004")))
```

Figure 5-4: Recipient Rule using LAST_NAME Attribute

everything about reservations that have been created, and this knowledge can be made available to us through attributes. As an example, let's consider a new attribute named TOTAL_MINUTES that only exists in a Sign Up Rule context. When a student submits the request to create a reservation, the Scheduling Server knows what day he or she wants the reservation to begin on. The Scheduling Server looks through all of the reservations on that day and tallies up the minutes from all of the reservations held by the student on that day. This value is put into TOTAL_MINUTES. Once this attribute exists, it can be used by a teacher to limit how much time a student using a rule like (TOTAL_MINUTES LESS_THAN_OR_EQUAL_TO "90").

### 5.4.2 Redefining Scheduling Server Payloads

#### Including More Student Information

As we have defined them, the payload of a Scheduling Server ticket determines most of the information the Scheduling Server knows about an incoming user. If these payloads are augmented to include more information, then the Scheduling Server can create more attributes to be used in enforcing Recipient Rules. For example, if payloads contained the last name of a student and this were stored in the attribute LAST_NAME, then a Recipient Rule could be created as seen in Figure 5-4. This would only allow students whose last name begins with A through N to sign up for time before February 14th, 2004. Students with last names beginning with O through Z could sign up for time after February 14th.

```
((GROUP_ID EQUAL_TO "1.001")))
 OR
(((GROUP_ID EQUAL_TO "1.00") AND (DATE BEFORE "2/14/2004"))
```

Figure 5-5: Rule written by teacher in Groups 1.00, 1.001

#### Multiple Groups for Teachers

If a teacher belongs to more than one group and the Scheduling Server allows them to define Recipient Rules and Sign Up Rules for each of their groups, then we can empower a teacher to coordinate details between groups. For example, if a teacher was in both group 1.00 and group 1.001 they could specify a Recipient Rule that would allow one class to sign up for time whenever they wanted, but would limit another class so that they could only sign up for time until February 14th (see Figure 5-5).

### 5.4.3 Self Describing Rules

Sign Up Rules must be satisfied any time a student attempts to create a reservation. However, in our prototype students cannot see any of the rules associated with a Time Distribution. They can only learn of them by inference when they attempt to sign up for time and violate one. Even if they could see the rules associated with a Time Distribution, they are not likely to understand our rule language.

It would be useful if rules could be translated to English and displayed to the student. For example, as `"The maximum time allowed for a reservation is 90 minutes"`. A simple but possibly problematic solution to this would be to alter the definition of a Rule so that it contains this translation: `(DURATION LESS_THAN "90" "The maximum time allowed for a reservation is 90 minutes")`.

This is problematic because errors would be easy to make and hard to detect. For example, the text associated with the rule above is incorrect, the explanation should read `"The maximum time allowed for a reservation is 89 minutes"`. It may be useful to research a solution to this by writing code to parse Rules and translate them into meaningful statements.

## 5.5 Scheduling Evolution

At present, first come, first serve scheduling appears to be the only mechanism that lab implementers are interested in. However, there are clearly more efficient ways to distribute resources. A major hurdle is that lab implementers want to enable experiments, not support the next generation of scheduling algorithms. We recommend creating a reference implementation of a first come, first served Scheduling Server. However, we also recommend designing this Scheduling Server to support scheduling algorithms in a modular fashion. In the beginning, it will only have the first come, first served scheduling module. However, we can work to identify courses that face non-traditional scheduling challenges, and provide solutions to them by creating modular scheduling extensions. They will support themselves by using our reference implementation with a non-standard plug in. If we follow this path, a Lab Server administrator will be able to contact a Scheduling Server and not just choose what time he or she wants to make available, but how he or she wants it to be made available.

# Bibliography


[1] D. Zych, *Client / Service Broker API*, [Online Document], September 2003, Available HTTP: http://web.mit.edu/jedidiah/ilab/index.html

[2] J. Harward, *Service Broker / Lab Server API*, [Online Document], October 2003, Available HTTP: http://web.mit.edu/jedidiah/ilab/index.html

[3] J. Harward, D. Zych, *Service Broker Administrative API*, [Online Document], November 2003, Available HTTP: http://web.mit.edu/jedidiah/ilab/index.html

[4] M. Fowler, *UML Distilled*, Addison-Wesley, Second Edition, 1999.

[5] D. Talavera, *On-Line Laboratory for Remote Polymer Crystallization Experiments Using Optical Microscopy*, Masters Thesis, Massachusetts Institute of Technology, 2003.

[6] P. Nasser, *Remote Microscope for Polymer Crystallization Web Lab*, Masters of Engineering Thesis, Massachusetts Institute of Technology, 2002.

[7] A. Kuchling, *Internet Access to an Optical Microscope*, [Online Document], November 1998, Available HTTP: http://www.mems-exchange.org/software/microscope/publications/ipc7-abstract.html

[8] T. Berners-Lee, *Hypertext Transfer Protocol : HTTP/1.1*, [Online Document], Available HTTP: http://www.w3.org/Protocols/rfc2616/rfc2616.html

[9] D. Booth, *Web Services Architecture*, [Online Document], Available HTTP: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[10] N. Mitra, *Soap Version 1.2 Part 0: Primer"*, [Online Document], Available HTTP: http://www.w3.org/TR/2003/REC-soap12-part0-20030624/

[11] D. Booth, *Internet Calendaring and Scheduling Core Object Specification*, [Online Document], Available HTTP: http://www.ietf.org/rfc/rfc2445.txt/