The iLab Debugging Service Broker:

A Module for Facilitating Development of Online Laboratories

by

Abhra D. Haldar

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[June 2006]

May 27, 2006

Author_____

Department of Electrical Engineering and Computer Science

May 27, 2006

Certified by_____

Jesús A. del Alamo

Thesis Supervisor

Accepted by_____

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

The iLab Debugging Service Broker: A Module for Facilitating Development of Online Laboratories

by

Abhra Haldar

Submitted to the Department of Electrical Engineering and Computer Science
on January 18, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The iLab Shared Architecture specifies a systematic and consistent way to architect new online laboratories. Despite its usefulness, however, building and debugging new labs under this architecture is an arduous process. The primary reason behind this is the use of Web services for communication. No standard tools exist for debugging Web services, so this naturally adds to the system's complexity.

The *Debugging Service Broker,* a new implementation of the iLab Service Broker, has been designed to address this problem. Using this broker, developers can intercept and view every SOAP message that passes through this module. They can then pinpoint errors in these messages or even make instant modifications to test their changes. We also envision using this expanded broker to provide mock lab clients and servers that can be used to conduct tests of individual modules in isolation.

Thesis Supervisor: Professor Jesús A. del Alamo
Title: Professor of Electrical Engineering

Acknowledgments

I would first and foremost like to thank my family for always supporting me in my personal and academic endeavors. Their encouragement has enabled me to persevere even in times of extreme duress.

I would also like to thank the terrific research team that I worked with, led by Professor Jesús del Alamo. I greatly appreciate his patience and understanding during these busy two terms of research. His dedication to the iLab project also served as a personal inspiration. I would also like to thank Jim Hardison for his willingness to help, no matter the task.

Table of Contents

Chapter 1: Introduction

The iCampus iLabs project is an MIT-led initiative aimed at simplifying the development of online laboratories ("iLabs"). The notion of an iLab stems from the inefficiencies inherent in a standard class laboratory setting. Typically, there are simply too many students for too few pieces of hardware. This arrangement taxes both students, who usually have limited time to learn a specialized device interface, and class staff, who must maintain and administer each of these machines.

iLabs address these limitations through the use of the Internet. Here, students simply submit Web requests to run an experiment on a remote piece of hardware. The laboratory server maintains a queue of such requests and executes them sequentially. This allows for effective load-balancing on the hardware, since the student no longer needs dedicated access to a device. This also eases the burden on course staff, since they can centrally set up and maintain these remotely accessible machines [1]. It also addresses scalability concerns, since supporting more students does not require an additional hardware investment [3].

The overlying benefit of iLabs is that it allows a shift in focus. Instructors can now craft relevant experiments without worrying about hardware malfunction. Moreover, since they can control students' means of access to a device,

teachers can provide a Web-accessible interface that meets the educational need without drowning students in its complexity of operation.

iLabs also present obvious economic advantages [1]. Classes will need far fewer hardware setups to serve their students, thus freeing them to purchase more specialized or advanced devices. One can even imagine devices at one educational institution serving students at several others through the Web.

1.1 Evolution of iLab

The notion of an iLab was originally realized through the MIT Microelectronics Weblab, a classroom tool that enabled students to remotely conduct current-voltage measurements on a microelectronics device or a small circuit using a Java applet client [5]. The project underwent several iterations and has been successfully used in a large MIT device electronics class since 1998.

During these initial deployments, it became that clear that the ad-hoc distributed architecture of Microelectronics Weblab would not suffice as a generalized architecture for future iLabs. As a result, in 2002, the iLabs team specified the *iLabs Shared Architecture* [2], which established modular units of lab functionality: the *lab client, lab server*, and *Service Broker*. The final unit acts a middleman between the client and server and also encapsulates much of the generic functionality that all iLabs typically share [8]. The team also published

XML (Extensible Markup Language) Web services [9] method descriptions for the interfaces between each of these units.

1.2 The Debugging Service Broker

In spite of this new architecture, which was aimed at easing lab development, it was discovered that the creation of new iLabs was still an arduous process. This was primarily due to the new architecture's use of Web services technology. The three disparate modules could communicate using Web services, but when a problem arose, the transmitted SOAP (Simple Object Access Protocol) [7] message was invisible and inaccessible to the developer. Given the multi-tiered architecture, developers need to follow the entire path of a raw SOAP transmission to pinpoint problem areas and debug effectively.

Another significant problem is the need to concurrently develop lab client and server. More specifically, without at least a barebones lab server, a developer cannot test out new feature changes on his client and vice versa.

The *iLab Debugging Service Broker* (DSB), an extension of the standard Service Broker, addresses these problems. The DSB offers one additional capability: interception of SOAP messages at any point along the architecture. Since all communication flows between the Service Broker and another module, the DSB adds an *intercepting layer* around the broker to capture SOAP transmissions.

After retrieving the message, the user can view or edit it before choosing to send it to the original destination.

Furthermore, using this interception functionality we can solve the aforementioned problem of concurrent client-server development. We can create a "dummy lab server" that, just like a normal lab server, waits for a SOAP input message, intercepts it, and then allows the user to type a SOAP response from scratch. Similarly, a "dummy client" would let the user type SOAP output messages for the server and retrieve actual results from running on the hardware.

This enhancement is important for several reasons. First, it promises to save time for future iLabs developers. Those new to the team can use it to quickly get an understanding of how iLabs SOAP messages are constructed. Also, the ability to dynamically edit an intercepted message enables developers to quickly diagnose and fix potential problems. Second, the dummy modules allow iLabs developers to individually build up client or server without needing the other. More significantly, it lets us test generic client and server components (i.e. a new client graphical engine) without being limited by slow server performance or limited client functionality.

1.3 Overview of Thesis

Chapter 2 first describes the Shared Architecture and its mechanisms in more depth. It then illustrates the difficulties in developing new labs, particularly in debugging, and shows how the DSB resolves each of these difficulties. This chapter also examines a hypothetical experiment request with particular focus on the behavior of the DSB. Chapter 3 gives a detailed account of the implementation of the SOAP interception/editing feature of the DSB, analyzing both server-side and client-side behavior. Chapter 4 draws conclusions on the DSB and proposes further work to be done to make the module more useful for lab developers.

As of this writing (5/06), the interactive or sensor architectures have not yet been finalized, so this paper will only focus on the batched variety of online lab.

Chapter 2: iLab Shared Architecture and the Debugging Service Broker (DSB)

This chapter takes a closer look at the different components and specifications of the iLab Shared Architecture. The interaction among these modules is depicted in Figure 1 [2]. It also identifies the current bottlenecks in developing new iLabs and introduces the Debugging Service Broker (DSB), a tool to mitigate these difficulties.

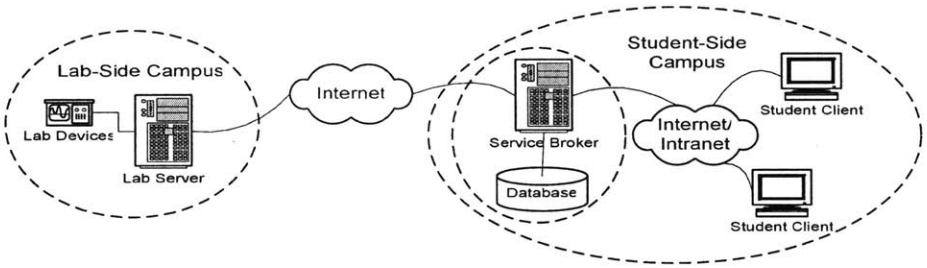


Figure 1: iLab Shared Architecture Components

2.1 The iLab Shared Architecture

The Shared Architecture specifies three different functional units. The lab client and server are holdovers from older ad-hoc approaches and allow the basic exchange of experiment specifications and results. The *Service Broker* is a new module encapsulating non-lab specific functionality, thus enabling iLab developers to concentrate on their specific domain.

2.1.1 Laboratory Client

The laboratory client is used to:

- Retrieve the laboratory configuration (including settable parameters)
- Send experiment specifications to the lab server
- View and analyze the resulting data

Java applets are the most commonly-used client technology. They have complete rich-client functionality and also have access to the Service Broker's credentials, which are necessary for lab server authentication. The main drawback of this technology is its potentially lengthy download time on each load. In addition, because Java applets have nearly monopolized this form of rich-client technology, developers must have a good understanding of this complex language. Even then, the task of building an applet client is an involved and time-intensive task.

2.1.2 Laboratory Server

The laboratory server is responsible for authorizing incoming requests from users and sending commands to the actual hardware of the laboratory. It typically manages a queue of incoming experiment requests. The server sequentially parses each experiment specification into a set of commands to dispatch to the hardware and waits for a response. This device data is subsequently processed

and converted into XML. The lab server then calls a Web service method, Notify(), to inform the broker of job completion.

2.1.3 Service Broker

The broker is the centerpiece of the iLab Shared Architecture. It serves several important functions, including:

- Authenticating/Authorizing users for lab server access
- Storing experiment records
- Launching client (applet or HTML)

The broker acts as a middleman between the client and the server. Users of the system log on to a service broker (authentication), launch an accessible client (permissioning), and subsequently perform client-lab server Web service requests transparently through the broker. [1] If authentication of the user is successful, the broker will add its credentials to the request so it is accepted by the lab server.

The benefit of the broker versus individual accounts is that one broker handles permissions for a multitude of client/server pairs. This entails that lab server developers do not have to concern themselves with individual user authorization;

[1] *Note:* For the remainder of this paper, one can assume that any contact between the lab client and lab server is invisibly transmitted through the Service Broker.

they may do it on a service broker basis. For instance, university B could centrally grant access to its lab server to all university A's registered students in certain class if it trusts A's broker [1].

2.1.4 Web services

Given that each of the three modules could run on a different computing platform, Web services seemed a natural mode of interaction. The Shared Architecture specifies detailed Web Services Description Language (WSDL) [6] interfaces for each module interface, in addition to several internal application programming interfaces (APIs) for Service Broker administrative functions.

The Shared Architecture also defines three XML schemas for sharing key information at certain points in the session.

- The *Lab Configuration* describes parameters about the laboratory setup that is relevant to the client, i.e. measurement parameters that must be set, available devices, etc. The client first makes a request for this document.
- The *Experiment Specification* is a description of the requested measurement created by the client. The lab server translates its contents into hardware commands and runs the measurement.
- The *Experiment Result* contains a data vector with the results of the previous measurement. The lab server forwards it back to the client.

2.2 An Example Experiment

The following section outlines a round-trip interaction with a batched experiment iLab built under the Shared Architecture.

1. A user logs on to a Service Broker at his college and selects a Service Broker group for the session.
2. He then launches an accessible lab client (determined by his group permissions) and receives the lab server's *Lab Configuration*.
3. The user prepares his experiment by setting the appropriate parameters in his lab client.
4. After the user submits a job, the client creates and sends the *Experiment Specification* to the Service Broker. The broker adds its own credentials to the message for Lab Server authorization. The client typically polls, calling RetrieveResult() periodically.
5. The Lab Server authorizes the incoming experiment request and then runs verification on the job. If successful, interprets the parameters, runs the experiment on the hardware, and constructs the *Experiment Result*. It then notifies the Service Broker of job completion.
6. The Service Broker may notify the client. The client now successfully polls for the *Experiment Result* and can present it to the user.

In summary, the Shared Architecture specifies three different modules and the Web service interfaces that are needed for transmitting experiment requests, results, and relevant messages.

2.3 iLab Development Methodology

The development of a new lab under the Shared Architecture is a straightforward process. Once the hardware has been arranged, the format of the lab configuration and experiment specification documents are specified. With this in hand, the client and the lab server can both be built up, either in sequence or in parallel.

The important thing to note is that either way, *one module cannot be tested* until we have a substantial part of the other functional. For instance, even if we built the client completely, we cannot verify its behavior until we have a basic lab server with the ability to interpret our experiment specifications and communicate with the hardware. Of course, this is not an advisable way to build a lab, since we want to modify each component incrementally after we test. This chapter will introduce a tool that will allow developers of iLabs to test each module in isolation. It accomplishes this by allowing users to impersonate either the lab client or server in a message exchange.

2.4 Debugging Web services

The iLab shared architecture leverages the platform independence of Web services to communicate among distributed systems. Yet debugging a Web services-based application is a frustrating task for the developer. For one, the error could lie in any or all of these distributed systems, perhaps even external to

the Web service producer and consumer. In addition, most Web services development environments shield programmers from the complexities of SOAP and help them adhere to an object-oriented programming paradigm. For instance, ASP.NET automatically marshals/unmarshals the relevant data fields from the SOAP message and passes them, as objects, to the Web service methods. Nowhere in the method lifecycle does the developer need to parse or write to the actual SOAP message body. Yet in debugging, developers need to follow the entire path of a raw SOAP transmission to pinpoint problem areas and debug effectively.

A generic *message tracer* would be a step in the right direction. Such a tool would forward each SOAP transmission to a third-party server, which would log the message for later reference. One product that provides this basic functionality is the now-deprecated MS Soap Trace Utility [10], although it suffers from dependence on the Windows platform. This is a particularly egregious limitation, given Web services' promises of platform independence.

Developers would also benefit from the ability to modify or even compose messages at different stages of the Web service invocation. For instance, perhaps a programmer notices a potential bug in the XML produced by the client; instead of fixing and rebuilding the client, he could fix the XML, forward the document, and see whether that change produces the right results. Another common scenario is when a Web service consumer has not yet been fully built,

but we still want to see the raw output produced by the Web service method. In such a scenario, a user could compose a SOAP message and impersonate the client by sending it to the lab server.

2.5 Solution: iLab Debugging Service Broker (DSB)

These problems in iLab development can be solved by using an *interception layer*. Simply put, this is a layer that sits in between each module (client and lab server) and the service broker. Incoming SOAP documents are stored here until the user decides to either confirm or modify the message. At that point, the SOAP message is forwarded to its original destination.

This concept is best understood through some common scenarios. The first scenario, depicting message passing in the standard SB, is provided for comparison purposes. In all the scenarios below, it is understood that a) every message sent between the lab client and lab server passes invisibly through the Service Broker, and b) users and their SBs have appropriate privileges on the target lab server.

2.5.1 Scenario 1: Standard SB Message Lifecycle[2]

In this arrangement, shown in Figure 2, a lab client is launched from a service broker. The client invokes a lab server method, which delivers a SOAP message to the broker (Step 1). The broker adds its own authentication information and forwards to the lab server, which authorizes and runs the experiment. The client polls and eventually retrieves the experiment results from the lab server (Steps 2, 3).

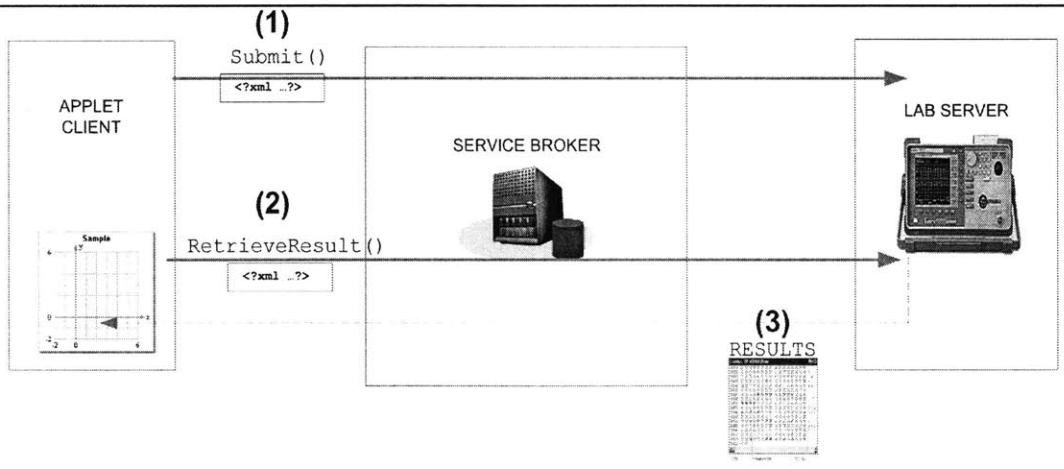


Figure 2: Sample method call in standard Service Broker

2.5.2 Scenario 2 (DSB): Trace Entire Message Lifecycle

In this scenario, shown in Figure 3 and Figure 4, a lab client is launched from a service broker. The client invokes a particular lab server method, which delivers

---

[2] *Note*: In the scenario diagrams, a solid line indicates a method request and a dotted, contiguous line indicates the corresponding response. The contents sent are positioned adjacent the sending module. The bold numbers represent stages in the message lifecycle, and are referenced in the text.

a SOAP message *X* to the interception layer (1). The user[3] is now shown this client-generated message and can choose to modify or forward as-is to the lab server (2, 3).

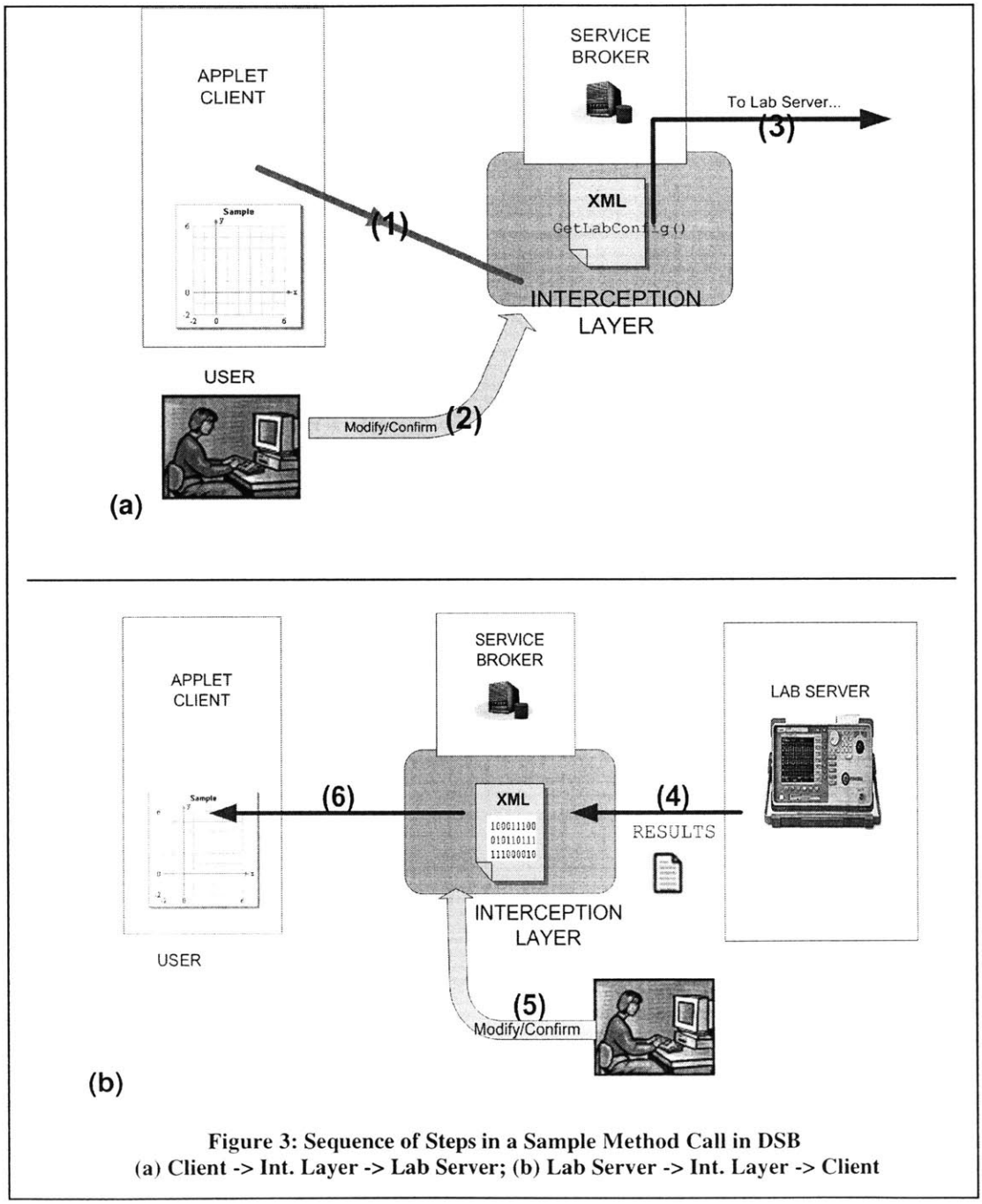


Figure 3: Sequence of Steps in a Sample Method Call in DSB
(a) Client -> Int. Layer -> Lab Server; (b) Lab Server -> Int. Layer -> Client

[3] It is understood that when using the DSB, the user and developer are the same. The terms will be used interchangeably henceforth.

The lab server generates a SOAP response *Y* to the interception layer (4). The user once again chooses to modify or accept (5), and the message is subsequently sent back to the client as the method response (6).

2.5.3 Scenario 3 (DSB): Testing Client w/ no Lab Server

For this testing scenario, illustrated in Figure 4, the DSB includes a 'dummy' lab server. The client issues a Web service request (1), which is modified and submitted by the user just as before (2, 3). However, in the absence of a lab server, the user enters a SOAP message as a response and sends it directly back to the client (4). There is no need to intercept this message on the response path because the user himself generated the message.

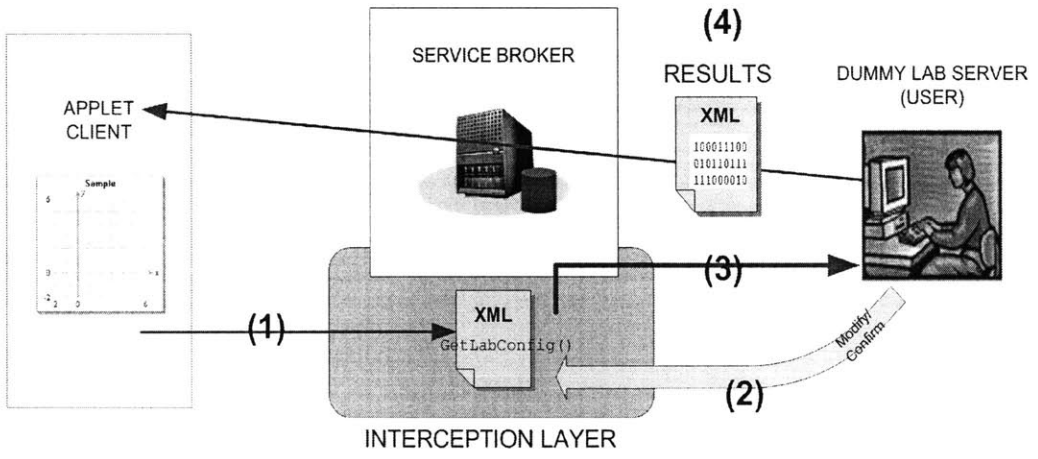


Figure 4: DSB allowing User to act as a "Dummy" Lab Server

Note that this is not limited to situations where no functional lab server exists. Suppose the lab client's graphical engine just underwent a major upgrade. Typically, users would test this by repeatedly sending requests to the lab server,

waiting for results, and examining the client graph. With this dummy lab server, the user can continually send pre-packaged lab server data that illustrates various features in the client applet. This avoids needless hardware execution wait time, since the user is interested in testing aspects of the client applet. In this approach, one can test each module efficiently and independently.

2.5.4 Scenario 4 (DSB): Testing Server w/ no Lab Client

For this testing scenario, the DSB includes a 'dummy' lab client. When launched, this client allows users to manually enter a SOAP message to send to a lab server (1). Here again, there is no need to capture the message in the interception layer since the user is creating it himself. The lab server generates the results and sends it to the interception layer (2). Since there is no client available to manipulate the data, the user would inspect the SOAP response directly to diagnose any potential lab server problems.

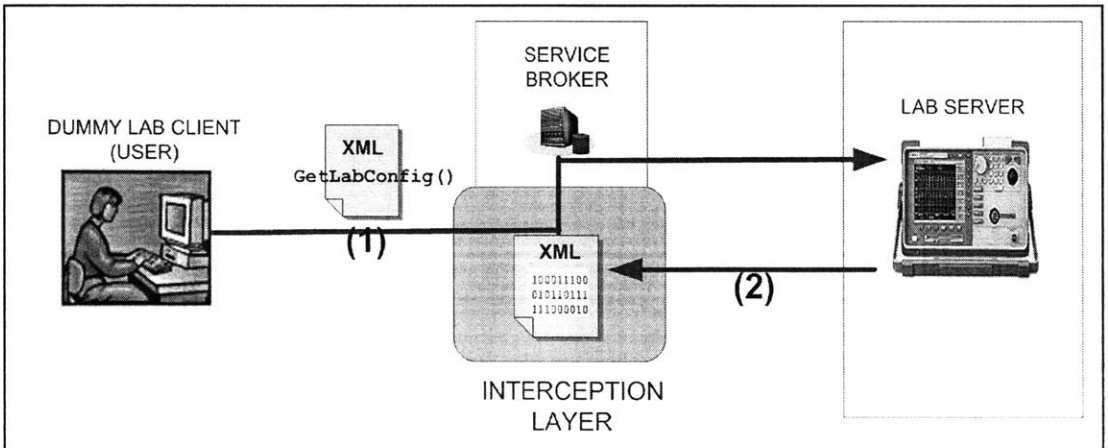


Figure 5: DSB allowing User to act as a "Dummy" Lab Client

2.6 Design Considerations

There were two options with regard to where to position the interception layer: a) on the same server as the service broker or b) on an external server. The pros and cons of each approach are considered below.

2.6.1 DSB-Embedded Interception Layer

In this model, the layer is located within the DSB itself. Thus, when a DSB Web service method is invoked, the method itself handles all necessary message storage and retrieval in the interception layer. In this scheme, the method directly interacts with this buffer through local method calls. This is markedly different from, for instance, sending networked requests to another server that implements the interception layer functionality. This design is illustrated in Figure 6.

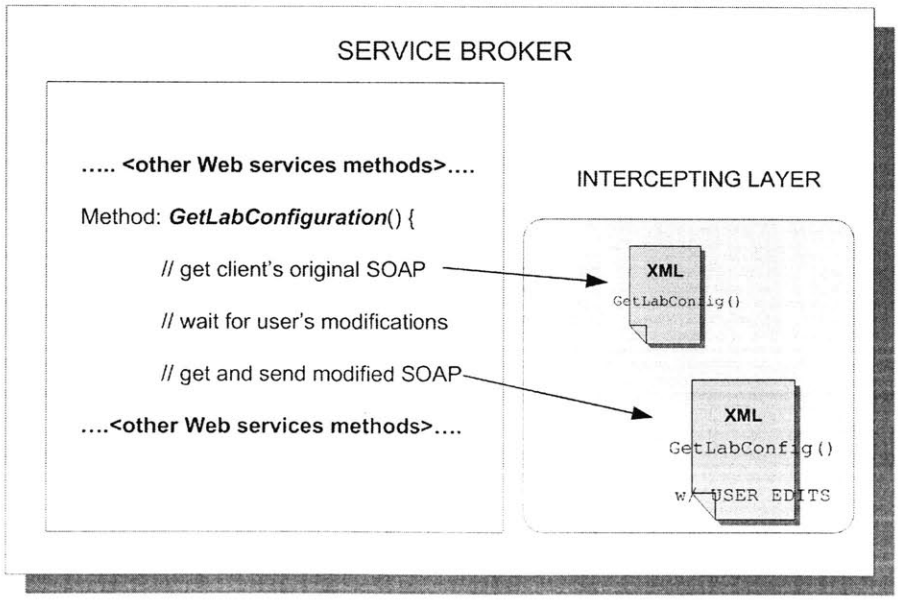


Figure 6: The DSB-Embedded Intercepting Layer Design

The advantage of this model is its conceptual synthesis of the functions of the DSB. The DSB, as the middleman in the iLab architecture, handles all message traces, modifications, and submissions in the system. It does not need to communicate with and rely on another server just for this interception functionality.

The disadvantage is primarily that this new functionality is now being tied into the SB's complex code base, potentially creating subtle new bugs. In addition, now that the two parts of the SB are more tightly coupled, maintenance and enhancements may also prove to be more difficult.

2.6.2 External Interception Layer

In this model, the layer is located on an external server, separate from the service broker. The user's client forwards its SOAP messages to this machine, which presents users with raw requests and responses for modification before sending to the ultimate destination. Unlike in the embedded model, here the DSB needs a way of communicating with this external server (for retrieving user-modified SOAP requests), and vice versa (for retrieving raw SOAP responses for user modification).

This scheme is depicted in Figure 7. The user retrieves the client's original SOAP request from the external server and makes any modifications. The DSB then retrieves these modifications (1) and sends the experiment for execution on the

lab server (2). The intercepting server then attempts to retrieve the experiment results from the DSB and present them to the user for final modification (3), before being forwarded to the client.

The main advantage of this approach is that the interception layer code is completely separate from the SB; in fact, no changes at all are required on the service broker's code base.

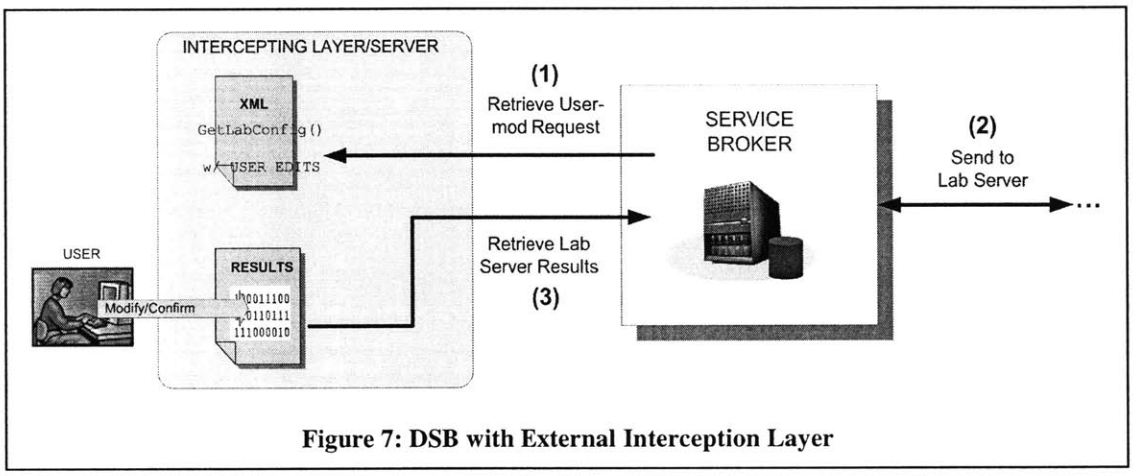


Figure 7: DSB with External Interception Layer

This is also the primary disadvantage, because to communicate with the service broker, the interception layer must programmatically tie into a service broker session (normally accomplished by user log in). This would require some inelegant code that would essentially "screen-scrape" SB credentials from the DSB login page. Evidently, this is poor from a maintenance point of view because it introduces a dependency on the system's site layout.

In the other direction, when the DSB attempts to retrieve intercepted messages from the external server, it must present some form of authentication as well.

This introduces the need for a separate SB authentication system on this external server, a non-trivial and unnecessary complication.

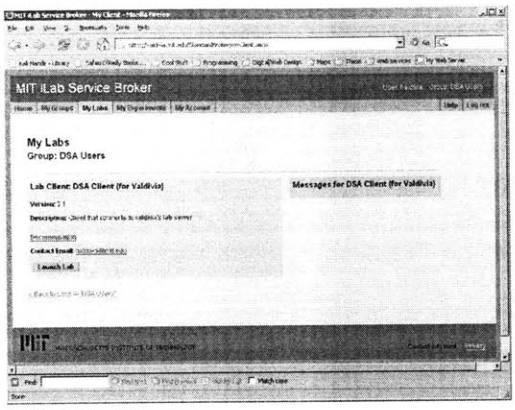
Ultimately, the DSB-Embedded Interception Layer is a better choice for this application and is the model adopted in my code. It cleanly ties together all DSB functionality and avoids the inelegant tethering of the external layer with the DSB.

2.7 Example: End-to-End Interaction

A step-by-step example of a complete message transaction (request, response, interception, and modification) best illustrates the potential benefits of the DSB. Screenshots of the actual DSB are presented here, although technical details about the implementation of the system are deferred until the next chapter. The purpose of this section is to concretely demonstrate the basic usage and operation of the DSB.

2.7.1 User Interface

Before delving into details, we compare the browser interfaces of the standard SB and the DSB in Figure 8. The obvious difference between the displays is the presence of an additional frame in the DSB, which we call the *interception window*. This frame serves as the user's main point of contact with the interception layer and is examined more carefully in Figure 9.



Standard Service Broker

Debugging Service Broker

Figure 8: Browser Interfaces for the Standard versus Debugging Service Broker

There are three main components in the interception window from Figure 9. (a) shows the request field, where SOAP requests from the client are presented to the user for modification. (b) shows the response field, where original SOAP responses are presented to the user for the same purpose before being forwarded back to the client. (c) will contain some log/timestamp data to help the user determine which message is being examined. This is currently not yet implemented, but will be useful in a later version of the DSB (see Chapter 4 for planned future work).

2.8 DSB Message Transaction - Walkthrough

This section gives a detailed walkthrough of a user's complete interaction with the DSB, starting from the client's generation of a request and ending at the receipt of a SOAP response. To further put this sequence of events in perspective, refer to Section 2.5 for a textual overview of the operation of standard SB. As mentioned previously, implementation details are omitted entirely in this section; they are presented in Chapter 3.

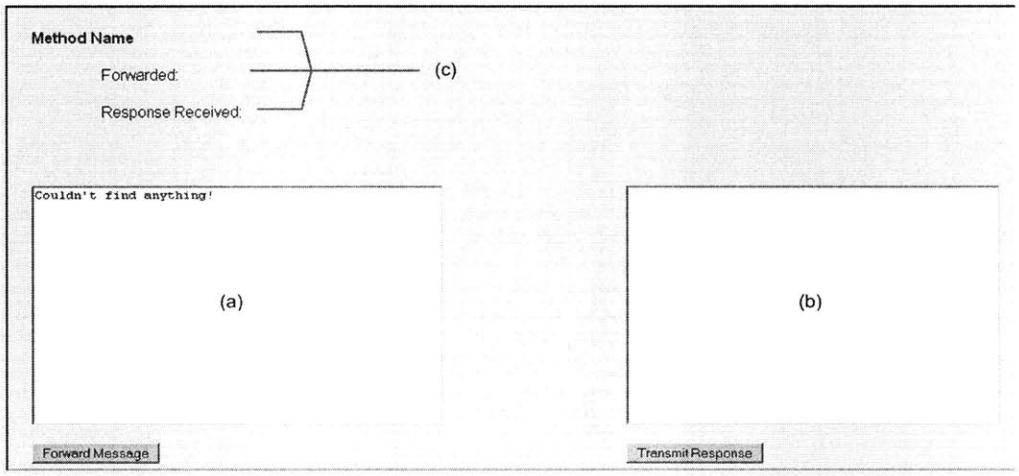


Figure 9: Structure of Intercepting Window
(a) Request Interception/Modification Field
(b) Response Interception/Modification Field
(c) Method Description/Timestamps

This walkthrough uses the client and server from the Dynamic Signal Analyzer (DSA) [11], a mature batched-experiment iLab. It also assumes the user has logged into the broker and that both the DSB and the user have the appropriate permissions for accessing the DSA lab server.

2.8.1 Step # 1: Request Method Interception

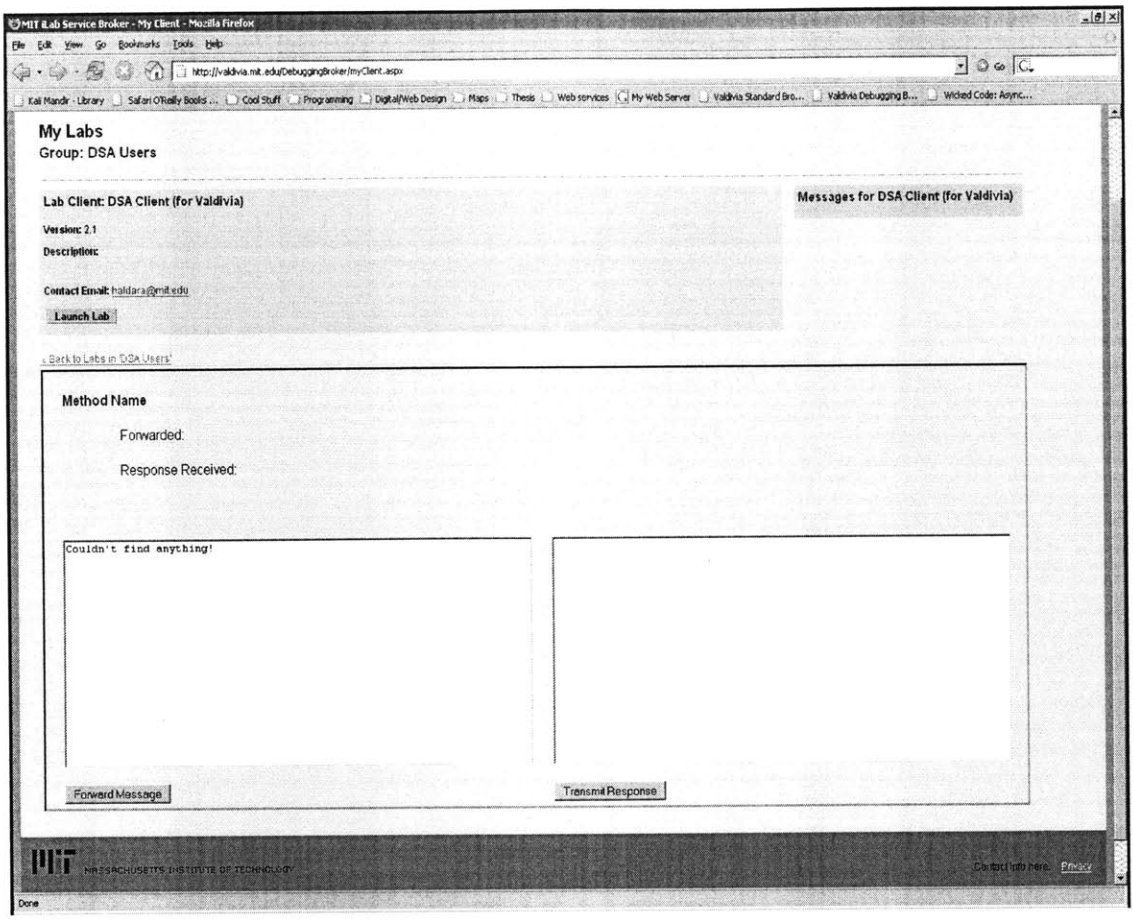


Figure 10: DSB polling for Client requests

The DSA client issues a *GetLabConfiguration()* request (to retrieve the lab configuration) before it displays itself. The interception window reloads itself periodically, checking for any new messages in the DSB's interception layer. It will continue to poll until it populates the request interception field on the left.

2.8.2 Step # 2: Modification of Intercepted Request

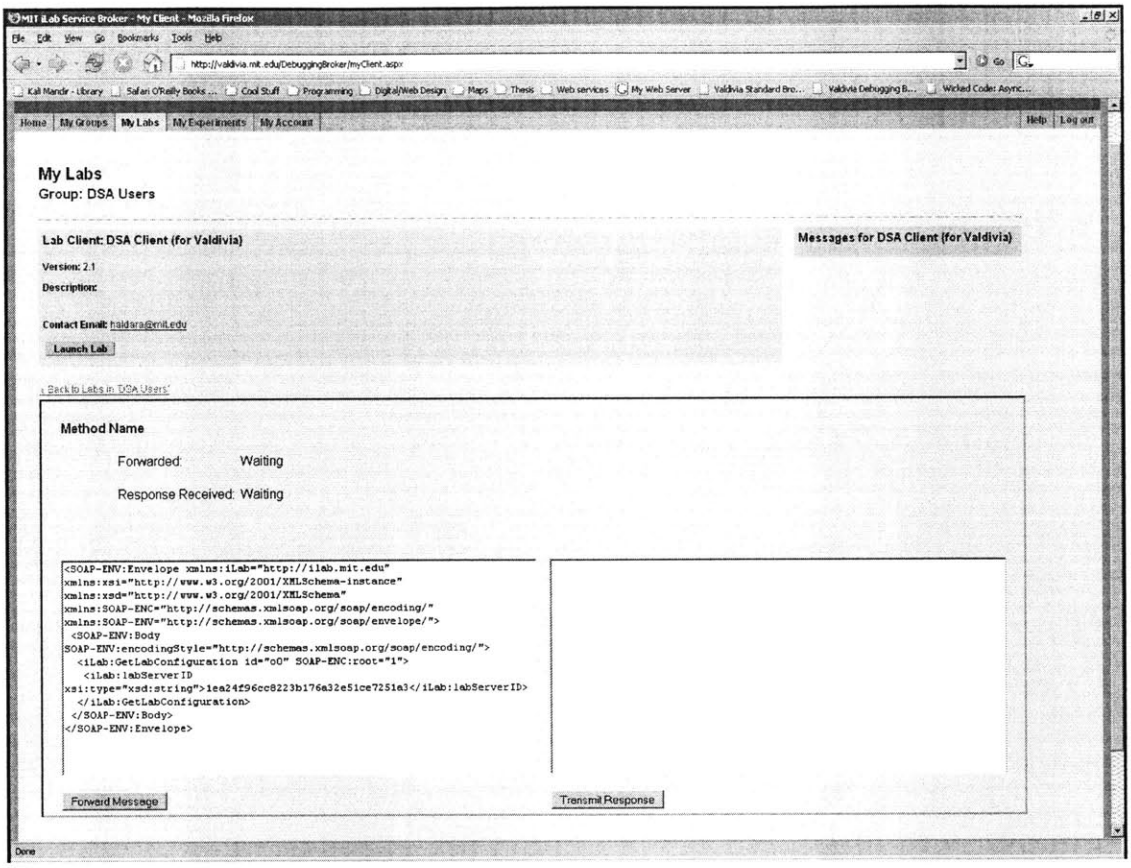


Figure 11: Intercepted SOAP Request; waiting for user modifications

Once the client's *GetLabConfiguration()* request has reached the interception layer, the window eventually finds it and populates the request field accordingly. At this point, reloading stalls while the DSB waits for the user to examine and modify the SOAP request as necessary. Once changes have been made, the user clicks "Forward Message" to forward the request to the DSB. The broker takes this modified message and transmits it directly to the Lab Server.

At this point, the interception window again begins to poll, now looking for the SOAP response from the Lab Server.

2.8.3 Step # 3: Intercepted SOAP Response

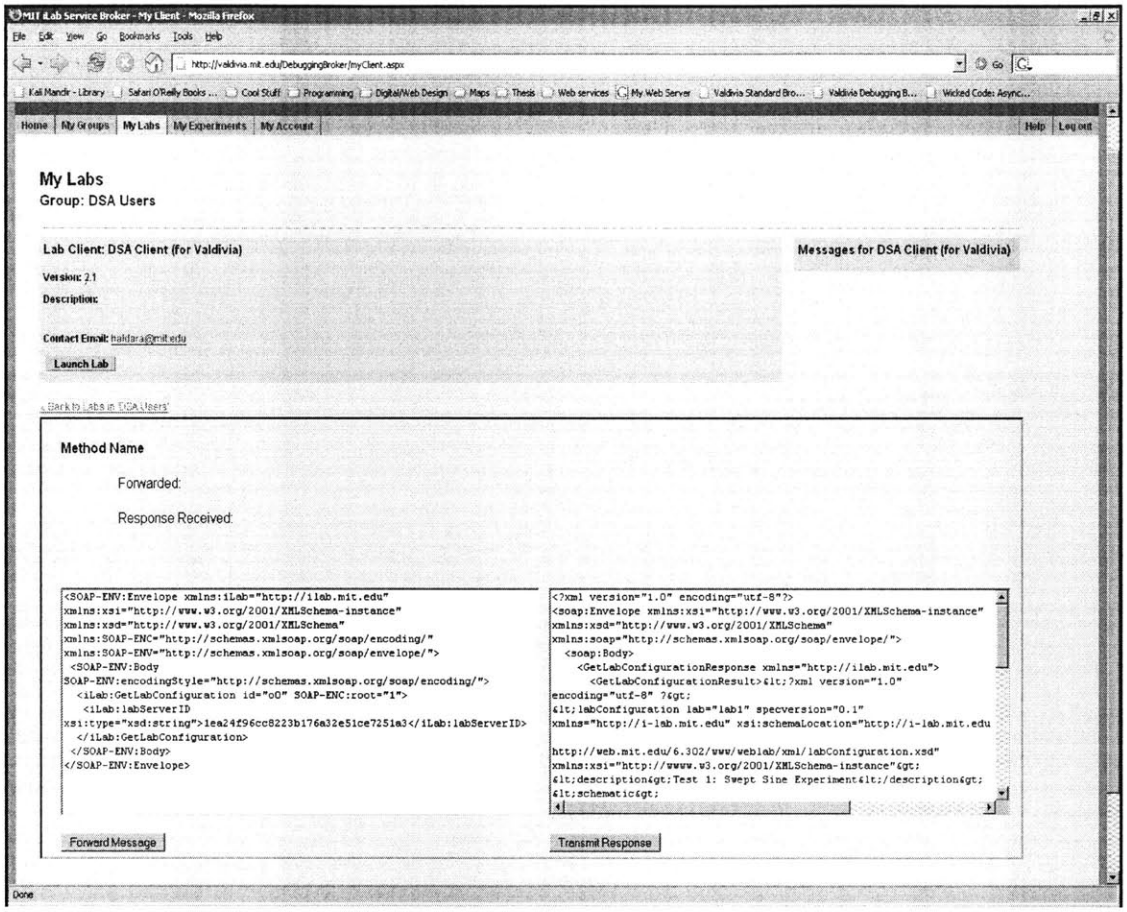


Figure 12: Intercepted SOAP Response from Lab Server; waiting for user modifications

The Lab Server processes the user-modified SOAP request and sends its response to the interception layer.

The interception window keeps polling until it retrieves this response, which is copied into the response text field. The window now stalls to allow the user to make his modifications to the response and click the "Transmit Response" button.

2.8.4 Step # 4: Forwarding SOAP Response to Client

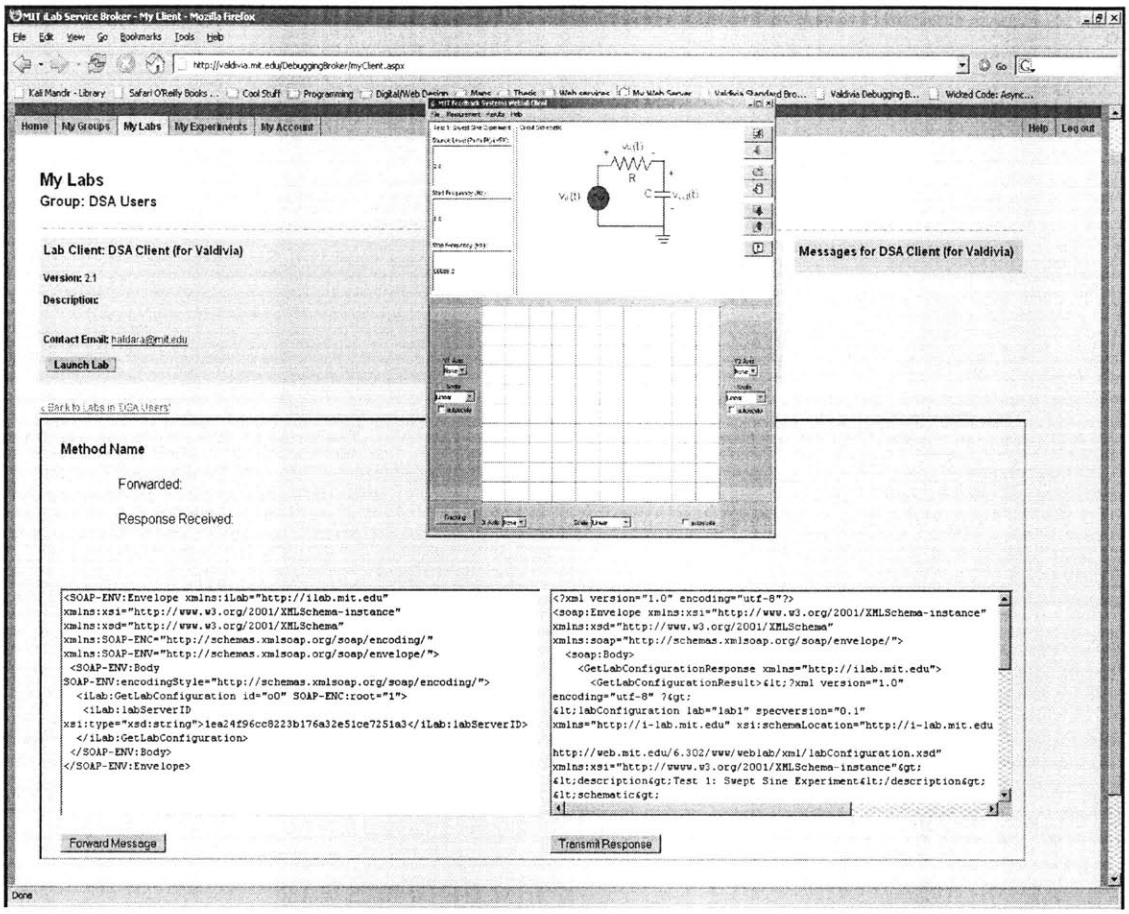


Figure 13: Forwarding of modified SOAP Response to Client

Once the developer has modified the original Lab Server SOAP response, the DSB retrieves this modified message and delivers it to the lab client as its response.

Upon receiving this response, the client constructs its interface (namely, parts that are dependent upon the lab configuration) and displays itself.

The next chapter will discuss the design of the DSB and notable technical successes achieved during the process. It describes in detail the implementation of each component in the DSB, and also the hurdles encountered in the course of development.

Chapter 3: System Architecture

This chapter takes an in-depth look at the low-level implementation of the DSB. Previous sections have described DSB user interaction via screenshots and diagrams. Here, the focus will be on

- outlining technical challenges faced during implementation and their respective solutions, and
- giving future developers a solid architectural overview to facilitate maintenance and build improvements upon the existing DSB.

3.1 Configuration

The DSB system is running on a Dell PowerEdge 650 server operating on Windows 2003. The server runs the iLabs v6.0 service broker release (with an embedded interception layer) on IIS 6.0 with a SQL Server 2003 backend.

To test the interception layer, we use the standard Dynamic Signal Analyzer Java client to contact a development copy of the DSA Lab Server. Our test DSB (henceforth named *Valdivia broker)* is running on the same machine [12]. Hosting both on one server is certainly not a requirement, and there is no reason why this arrangement should affect the behavior of either module.

3.2 Overview

We first sketch, in some technical detail, the standard sequence of events when a user's client makes a Web service request through the DSB. For a more comprehensive, high-level overview with screenshots, refer to Chapter 3, Section 5 or Figure 3.

3.2.1 Request Stage (Client -> Interception Layer -> Lab Server):

This section walks through a sample Web service client request. It examines the interplay between the user, interception window, and DSB in transmitting an experiment specification to the Lab Server. The next section does a similar analysis for the Lab Server response back to the client.

1. Client launched => triggers a Web service call (***WSCall()***) on the DSB (sends SOAP Request ***R***)
2. User browser's "interception window" (Chapter 3, Section 5.1) periodically polls database for newly intercepted messages
3. Web service receives ***R***, records it in a database table of traced messages
    a. Method stalls until it detects user confirmation/modification of ***R***
4. Once window finds ***R***, it is presented to user for modifications (***R -> R'***) and forwarding
5. ***R'*** retrieved by DSB, sent as input to Lab Server
    a. Web service blocks while experiment executes
    b. User's interception window polls for SOAP response

3.2.2 Response Stage (Lab Server-> Interception Layer-> Client):

A similar process occurs after the Lab Server generates a response ***S.***

- DSB Web service method inserts ***S*** into the traced messages table, stalls until it detects confirmation/modification of ***S***
- User's interception window retrieves ***S,*** approves/makes changes (***S -> S'***)
- Web service returns ***S'*** as SOAP output to the client

3.3 Detailed System Design

This section first looks at the intricacies of handling Web services in ASP.NET, the environment in which the DSB runs. It describes shortcomings in standard ASP.NET Web services processing and proposes an architectural solution to resolve these difficulties.

The remainder of this section also looks at the implementation of the browser interception window and examines a challenging concurrency issue that shaped the design of this module.

3.3.1 ASP.NET Web services processing

ASP.NET offers a great deal of infrastructure for using existing Web services. WSDL descriptions of a Web service method can be automatically generated from any .NET method, so a third party can easily construct the appropriate SOAP envelope in sending a request.

ASP.NET also provides marshaling/unmarshaling features for its Web services methods, extracting the necessary argument values from the SOAP request and appropriately packaging the return values in the SOAP response. With this paradigm in place, Web service methods, and thus .NET developers, typically do not have to manipulate messages on an XML level.

3.3.2 The *SoapExtension* Class

With this ease of use comes a price – ASP.NET Web services do not, by default, allow methods access to the underlying SOAP messages. Yet for interception and debugging, this is an absolutely crucial requirement since users are manipulating request/response messages on an XML level.

Fortunately, ASP.NET provides a supplemental facility for lower-level processing [13]. The *SoapExtension* class exposes the SOAP input and output streams, typically hidden by the ASP.NET infrastructure, to any associated Web services method. It grants these methods access to the raw XML of the SOAP request and the ability to manually write XML into the SOAP response. The class' `ProcessMessage()` function is called during each stage of the Web service method lifecycle, specifically before and after marshaling/unmarshaling. Listing 1 demonstrates, via informal pseudocode, the general structure and function of `ProcessMessage`() in the DSB when associated with a Web service procedure called *methodY()*.

SoapExtension for methodY()
ProcessMessage() pseudocode

```
InputStream soapRequest;
OutputStream soapResponse;

Database db;

void ProcessMessage()
{
      // original request => DB
      store_client_request_db(soapRequest.contents)

      // poll for user modified request
      while(!db.hasModifiedRequest())
            // sleep for three seconds and poll again

      // makes the modified request available to WS method
      soapRequest.contents = db.getModifiedRequest()

      // generates LS result and stores in db
      methodY()

      // poll for user modified response
      while(!db.hasModifiedResponse())
            // sleep for three seconds and poll again

      // send modified response as WS method output
      soapResponse.contents = db.getModifiedResponse()

}
```

Listing 1: Pseudocode for `ProcessMessage()` - Role in DSB

Figure 13 shows how *SoapExtension* fits into DSB by tracing through a *GetLabConfiguration()* request from the client. Upon receiving a request from the input stream, the *SoapExtension* for *GetLabConfiguration()* stores the message in a local database table *Traced_Messages* (see Appendix A for table structure) and stalls, polling for user input (1). The database table essentially acts as the interception layer buffer described previously. In its interception window, the user's browser retrieves this message directly from the database and permits any modifications (2). Once the user has submitted any changes, the *SoapExtension* finds the modified message and passes it as input to the main body of *GetLabConfiguration()* (3). In the meantime, the *SoapExtension*

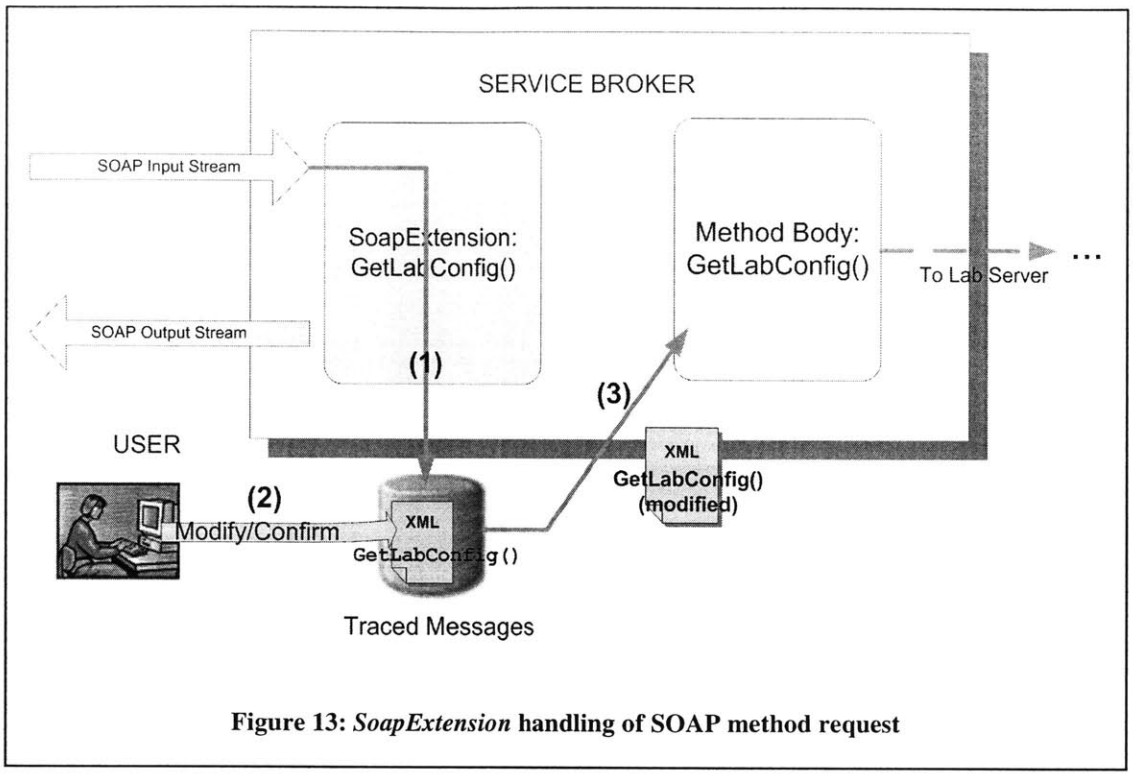


Figure 13: *SoapExtension* handling of SOAP method request

now begins polling for a modified Lab Server response (see Figure 14). This method can now parse the input as necessary before executing the experiment on the lab server. Figure 14 illustrates the role of *SoapExtension* on the response path. The results are retrieved by *GetLabConfiguration()*'s method call, upon which they are delivered to the `Traced_Messages` database table (1). Once the user's interception window has retrieved this results message and he has made his changes (2), *SoapExtension* finally stops polling and copies the modified response to the SOAP output stream (3). Whatever message is copied to the

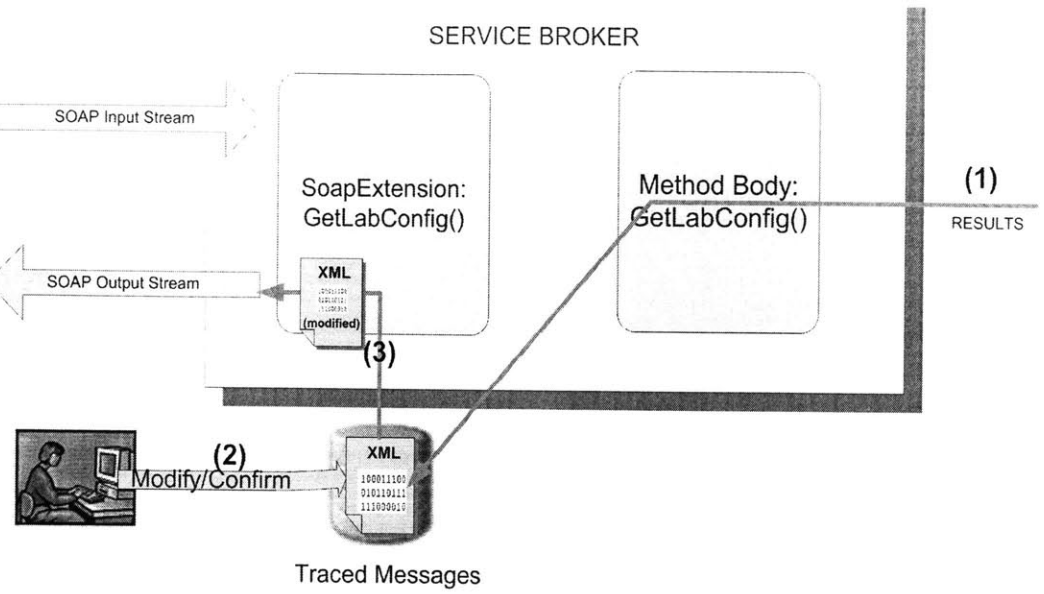


Figure 14: *SoapExtension* handling of SOAP method response

output stream will then be sent back to the originator of the request, the lab client.

This new model entails a significant change in the way a Web service method behaves. Previously, the method processed its arguments (demarshaled from the SOAP input) and returned an object (marshaled into the SOAP output). With a *SoapExtension*, the method can still use its demarshaled arguments, but since the extension is expecting its response on the output stream, the service method must write its XML response to the output stream. This allows `ProcessMessage()` to subsequently operate on its contents.

Appendix B contains a simplified listing of the modified *GetLabConfiguration()* method and its associated *SoapExtension*.

3.4 Client Side Behavior

As in the standard SB, the user launches the client through the page `myClient.aspx` (Figure 8). In the DSB, as explored in Chapter 3, this page contains an additional *interception window* called `myMessages.aspx` (Figure 9). This page shows users' traced SOAP messages and allows them to modify and send them back to the appropriate *SoapExtension*.

`myMessages.aspx` initially reloads itself periodically, searching the database for a Web service request issued from the applet. Once a message was found and displayed to the user, the page would stop reloading and wait for the user to submit changes to the database; the Web service retrieves and uses this as its input message. `myMessages.aspx` subsequently starts reloading itself, searching the database now for the SOAP response message. When this message is found and displayed to the user, the page stops reloading and allows the user to enter his changes. This changed copy is finally submitted, first to the DSB and then to the client.

3.4.1 Reloading Conditions

This page is set to reload only when it expects to read a message from the database: either the original SOAP request or the unmodified SOAP response. `myMessages.aspx`' server-side code checks for these two conditions and if met, programmatically inserts the necessary reloading JavaScript into the page.

The server-side code for `myMessages.aspx`' is given in Appendix C.

3.4.2 IIS Concurrency Problems

Microsoft Internet Information Server (IIS) 6.0, a popular Web server supporting ASP.NET, typically assigns a dedicated request thread per current session. Thus, if a user tries to launch two dynamic pages that require processing on the server, the assigned thread will service one request and block on the other.

This property of IIS led to a subtle threading issue in `myClient.aspx`. The initial version of `myMessages.aspx` had been part of the session (by default). However, because the Web service methods sent by the applet are implicitly requested through `myClient.aspx`, `myMessages.aspx`' reload call blocks while the Web service invocation completes.

We explored two potential solutions to this problem. One way to achieve concurrency without explicitly using more threads is to use *asynchronous programming*. In .NET, one can dispatch a job to be performed concurrently and automatically invoke a callback method once it completes. We could theoretically have the database message polling performed asynchronously and in the callback, update the display when complete. This would not work, however, because the callback executes on the server and cannot affect the browser display.

The other solution to this problem is to disable state on one of the concurrent pages, a feasible yet inelegant workaround. Thus, IIS can dispatch `myMessages.aspx` and `myClient.aspx` to separate request threads since the one thread/session constraint no longer applies. Yet with this approach, we must explicitly pass the iLab session identifier to the page as an HTTP request parameter. This value, typically stored as a cookie, is used by `myMessages.aspx` to identify messages from the current session.

3.5 Technical Evaluation

Building a system connecting so many disparate parts together is a daunting task, especially when the interactions between them are hard to systematically capture. Nevertheless, the current incarnation of the DSB, while still very much a prototype, manages to fulfill its original objectives and in a clean way.

The following chapter will draw specific conclusions about the success and efficacy of the DSB approach. It will evaluate the system on various standard metrics, such as extensibility and ease of maintenance. This section will also outline the work that needs to be done before the DSB can be shifted to a production setting.

Chapter 4: Conclusions and Recommendations for Future Work

4.1 Progress on Objectives

My overall impression of the DSB was very positive. The system was architected in a conceptually clear manner with definite end goals in mind, and those objectives were all met by the system. Most notably, the DSB supports each of the usage scenarios illustrated in Section 2.5.

4.2 Evaluation of System Design

The following section critiques the DSB on the basis of its architectural merits.

4.2.1 Modularity and Maintainability

Despite the assertion that binding the intercepting layer and SB code together would hurt maintainability, the *SoapExtension* mechanism allows a nice separation between Web services method logic and logic to control the method's SOAP input and output. Because of this conceptual divide, we have two modular components. Most importantly, because the Web services methods never directly know of their *SoapExtension*s, we introduce only very limited changes into the method bodies.

4.2.2 Extensibility

The *SoapExtension* approach also proved very extensible. Because `ProcessMessage`() does not contain any code specific to a given Web services method, our lone *SoapExtension* class can potentially handle all the Web services methods in the Shared Architecture. Currently, it is associated with all the major method calls, such as *GetLabConfiguration(), Submit(),* and *RetrieveResult()*.

4.2.3 Usability

As demonstrated by the screenshots in the last chapter, the DSB's interception window is easy-to-use and convenient since it is launched within the Service Broker page itself. The reloading behavior was designed to be non-intrusive (i.e. no reloading while user is modifying a SOAP message) and works just as expected.

4.3 Recommendations for Future Work

Although the current version of the DSB is a successful proof-of-concept, there are a few additional pieces of functionality that can be added to increase its overall usefulness.

4.3.1 Grant User Greater Control

One problem with the current design of the system is that the user has very little control over the flow of execution in the system. Regardless of what feature or problem he is debugging, the user must passively intercept and forward all messages that pass through the system.

One idea for improving the user experience is to grant him greater control over the interception behavior. For instance, if the developer knows the problem lies in retrieving results, there is no need for him to capture and view *GetLabConfiguration()*. This can be accommodated by dynamically allowing users to select the methods they want to intercept during a session – in a checkbox within the interception window, for instance.

Another feature that would help the user is a dynamically maintained log of all method calls during a particular session. The interception window might show a table that lists all messages, times sent, etc. Selecting a particular entry within the table would populate the request and response fields with the corresponding SOAP messages from that method call (not editable, of course). The user can, for example, try a variety of different modifications to a request and then compare each of their responses, leading to more effective debugging.

All in all, this project was a terrific opportunity to address some of major inefficiencies in the iLab development process. The DSB has so far shown

promising results. With an increased emphasis on the user experience, it has the potential to become an invaluable tool for the lab developer. It is also my sincere hope that a streamlined debugging system such as the DSB will decrease the technical barriers to entry posed by the distributed iLab architecture and Web services technology.

Appendix A -- Interception Layer: DB Table

The interception layer is accessed asynchronously by the various DSB components. The layer, which acts as a buffer for incoming messages until developers retrieve and modify them, is actually implemented as a table in a database.

The structure of the table, `Traced_Messages`, is as follows:

Column Name	Data Type	Description
Message_id (= primary key)	Int	Unique id for each message
User_id	Varchar (500)	Cookie identifier unique per session
Orig_message	Varchar (2000)	SOAP request sent by client
Mod_message	Varchar (2000)	Orig_message modified by user
Response	Text (16)	SOAP response received from Lab Server
Response_modified	Text (16)	Response modified by user
Message_received	Datetime	Time Orig_message received by DSB
Mod_message_received	Datetime	Time Mod_message submitted by user
Response_received	Datetime	Time Response received by DSB from Lab Server
Complete	Smallint (0,1)	Indicates whether response has been transmitted to client

Appendix B – Sample *SoapExtension*

To allow users to access raw SOAP messages in ASP.NET, a Web service method must have an associated *SoapExtension* subclass. The bulk of the work in this class is done by the `Process_Message()`. A sample implementation of this method, simplified from the actual DSB code, is provided below for illustration purposes.

```
[MethodImpl(MethodImplOptions.Synchronized)]
public override void ProcessMessage(SoapMessage message)
{
      try
      {
            switch (message.Stage) {

                  case SoapMessageStage.AfterDeserialize :
            {


                  DateTime currTime = DateTime.Now;
                  double millis = 0;

                  // loop until we find a modified message
                  while(millis < 100000)
                  {
                       string command = "SELECT mod_message FROM
                                 TRACED_MESSAGES WHERE message_id = " +
                                 "(SELECT MAX(message_id) FROM
                                 TRACED_MESSAGES WHERE user_id = '" +
                                 context.Request.Cookies["iLabSBCookie"].
                                 Value + "')";

                        SqlCommand comm =
                              new SqlCommand(command, myConnection);

                        SqlDataReader readSQL = comm.ExecuteReader();

                        readSQL.Read();


                        if(readSQL.IsDBNull(0))
                        {
                              System.Threading.Thread.Sleep(1000);
                              continue;
```

```
            }

            // recovered a mod message
            string modMessage = readSQL.GetString(0);

            readSQL.Close();
            break;
        }

        HttpContext.Current.Request.InputStream.Position = 0;


        HttpContext.Current.Items["SoapInputStream"] = modM

        StreamReader rdr = new
              StreamReader(context.Request.InputStream);
        StringWriter writer = new StringWriter(modMessage);
        writer.Write(rdr.ReadToEnd());

        HttpContext.Current.Items["SoapOutputStream"] =
                                  appOutputStream;

        break;
}

case SoapMessageStage.AfterSerialize :
{

        StreamReader reader = new StreamReader(appOutputStream);


        SqlConnection myConnection = new SqlConnection(
              System.Configuration.ConfigurationSettings.
              AppSettings["conn"]);


        myConnection.Open();

        string command =
              "UPDATE TRACED_MESSAGES SET response = '" + messages
              + "', response_received = GETDATE() WHERE message_id
                          (SELECT max(message_id) FROM
                          traced_messages WHERE " +
                            " user_id = '" +
                          HttpContext.Current.Request.Cookies["iLab
                          SBCookie"].Value + "')";

        SqlCommand sqlComm = new SqlCommand(command, myConnection);

        sqlComm.ExecuteNonQuery();

        myConnection.Close();

        // poll for the modified response message to come in
        while(millis < 100000)
        {
           string command = "SELECT response_modified FROM
                    TRACED_MESSAGES WHERE message_id = " +
                    "(SELECT MAX(message_id) FROM
                    TRACED_MESSAGES WHERE user_id = '" +
```

```
                                context.Request.Cookies["iLabSBCookie"].V
                                alue + "')";

                                SqlCommand comm = new SqlCommand(command,
myConnection);

                                SqlDataReader readSQL =
comm.ExecuteReader();

                                readSQL.Read();


                              if(readSQL.IsDBNull(0))
                              {
                                  System.Threading.Thread.Sleep(1000);
                                  Continue;
                              }

                            TimeSpan ts = DateTime.Now - currTime;

                            millis = ts.TotalMilliseconds;

                            readSQL.Close();

                            continue;
                      }

                      else
                      {
                            // found modified response
                            string modResponse = readSQL.GetString(0);

                            readSQL.Close();
                            break;
                      }
                }



                      // copy this output to the output stream
                      StreamReader rdr = new
                            StreamReader(context.Request.InputStream);
                      StringWriter writer = new StringWriter(modResponse);
                      writer.Write(rdr.ReadToEnd());


                      //CopyStream(appOutputStream, httpOutputStream);
                      strm.Flush();
                      strm.Position = 0;
                      CopyStream(strm, httpOutputStream);

                      strm.Close();
                      }
                }
```

Appendix C -- myMessages.aspx

The user's interception window in the browser is generated by myMessages.aspx. The following C# code excerpt details the loading of this window:

```
public class myMessages : System.Web.UI.Page

        {
                private void Page_Load(object sender, System.EventArgs e)
                {
                        if (Page.IsPostBack)
                        {
                                // add a javascript block
                                string newPage = "myMessages.aspx?id=" +
HttpContext.Current.Request.QueryString["id"];
                                string jScript = @"<script language='javascript'>InitializeTimer('"
+
                                                newPage + "');</script>";
                                Page.RegisterClientScriptBlock("PopWindow6",jScript);

                                return;
                        }

                        // session state disabled, so have to pass this manually
                        ilabID = HttpContext.Current.Request.QueryString["id"];

                        myConnection.Open();

                        DateTime currTime = DateTime.Now;

                        //string ilabID = Request.QueryString("id");
                        //sw.WriteLine("ilab ID: " + ilabID);

                        try
                        {
                                string command = "Select message_id, orig_message,
response, mod_message, message_received, mod_message_received, response_received " +
                                        " FROM Traced_Messages WHERE message_id = " +
                                        "(SELECT max(message_id) FROM
TRACED_MESSAGES WHERE complete = 0 AND user_id = '" +
HttpContext.Current.Request.QueryString["id"] + "')";
```

```
                    SqlCommand sqlComm = new SqlCommand(command,
myConnection);

                    SqlDataReader reader =
                        sqlComm.ExecuteReader();

                    // if there is any such message
                    if(reader.Read())
                    {
                        string mod_message;
                        //string orig_message;
                        string response_message;

                        if (!reader.IsDBNull(2))
                        {
                            sw.WriteLine(DateTime.Now + ": Retrieved a
response...");

                            // everything else should be available
                            response_message = reader.GetString(2);
                            //orig_message = reader.GetString(1);
                            mod_message = reader.GetString(3);

                            responseBox.Text = response_message;
                            messageBox.Text = mod_message;

                            //DateTime dt = reader.GetDateTime(6);
                            //lblResponseTime.Text = dt.ToString();

                        }

                        // otherwise, just check whether the modified message
has gotten in yet
                        else
                        {
                            //lblResponseTime.ForeColor = Color.Green;
                            lblResponseTime.Text = "Waiting";

                            // modified message has been received, but no
response
                            if (!reader.IsDBNull(3))
                            {
                                sw.WriteLine(DateTime.Now + ":
Retrieved the modified message only...");
                                messageBox.Text =
reader.GetString(3);

                                // extract the time this modified message
was sent
                                DateTime dt2 = reader.GetDateTime(5);

                                lblForwardTime.Text = dt2.ToString();

                                string newPage =
"myMessages.aspx?id=" + HttpContext.Current.Request.QueryString["id"];
```

```
                                                    string jScript = @"<script
language='javascript'>InitializeTimer('" +
                                                            newPage + "');</script>";
//                                                  string jScript = @"<script
language='javascript'>InitializeTimer(""http://www.cnn.com/"");</script>";

        Page.RegisterStartupScript("PopWindow2",jScript);
                                                }

                                                // only original message is retrieved
                                                else
                                                {
                                                        lblForwardTime.Text = "Waiting";

                                                        sw.WriteLine(DateTime.Now + ":
Retrieved the original message...");
                                                        messageBox.Text =
reader.GetString(1);
                                                }
                                        }
                                }

                                else
                                {
                                        string newPage = "myMessages.aspx?id=" +
HttpContext.Current.Request.QueryString["id"];

                                        string jScript = @"<script
language='javascript'>InitializeTimer('" +
                                                newPage +
"');</script>";Page.RegisterStartupScript("PopWindow3",jScript);
                                }

                                reader.Close();
                                myConnection.Close();
                                sw.Close();

                                return;

                }

                        catch (Exception e3)
                        {
                                sw.WriteLine(DateTime.Now + "********EXCEPTION
ENCOUNTERED*******: " + e3.ToString() + "\nTrace: " + e3.StackTrace);
                        }
                        finally
                        {
                                //sw.WriteLine(DateTime.Now + ": Outta here...");
                                myConnection.Close();
                                sw.Close();
                        }
```

Bibliography

1. "The Challenge of Building Internet Accessible Labs." 2001. Accessed 10 May 2006 <http://icampus.mit.edu/iLabs/architecture/downloads/downloadFile.aspx?id=2>.

2. V. Judson Harward, J. A. del Alamo, et al: "iLab: A Scalable Architecture for Sharing Online Experiments". *International Conference on Engineering Education* (Gainesville, FL, 16-21 October 2004).

3. J. Hardison, D. Zych, J. A. del Alamo, et al: "The Microelectronics WebLab 6.0 – An Implementation Using Web Services and the iLab Shared Architecture". *Exploring Innovation in Education and Research* (Tainan, Taiwan, 1-5 March 2005).

4. G. Viedma, I. Dancy, et al: "A Web-Based Linear-Systems iLab". 2005.

5. J. A. del Alamo, L. Brooks, et al: "The MIT Microelectronics WebLab: a Web-Enabled Remote Laboratory for Microelectronics Device Characterization". *2002 World Congress on Networked Learning in a Global Environment* (Berlin, Germany, May 2002).

6. E. Christensen, F. Curbera, et al: "Web Services Description Language (WSDL) 1.1". W3C. 2001. Accessed 11 May 2006. <http://www.w3.org/TR/wsdl>.

7. M. Gudgin, M. Hadley, et al: "SOAP Version 1.2 Part 1: Messaging Framework". W3C. 2003. Accessed 11 May 2006. <http://www.w3.org/TR/soap12-part1>.

8. K. Yehia: "The iLab Service Broker: a Software Infrastructure Providing Common Services in Support of Internet Accessible Laboratories", MIT Master of Science thesis, May, 2004.

9. D. Booth, H. Haas, et al: "Web Services Architecture". W3C. 2004. Accessed 9 May 2006. <http://www.w3.org/TR/ws-arch>.

10. "SOAP Toolkit 3.0". 2005. Accessed 10 May 2006. <http://www.microsoft.com/downloads/details.aspx?familyid=c943c0dd-ceec-4088-9753-86f052ec8450&displaylang=en#QuickInfoContainer>.

11. G. Viedma: “Design and Implementation of the Feedback Systems Web Laboratory”. Master of Engineering Thesis. MIT, 2005.

12. <http://valdivia.mit.edu/DebuggingBroker>

13. T. Ewald: “Accessing Raw SOAP Messages in ASP.NET Web Services”. 2003. Microsoft. Accessed 5 May 2006. <http://msdn.microsoft.com/msdnmag/issues/03/03/WebServices>