JAVA Implementation of the Batched iLab Shared Architecture

http://dx.doi.org/10.3991/ijoe.v9iS3.2531

L.J. Payne, M.F. Schulz
The University of Queensland, Brisbane, Australia

***Abstract*—The MIT iLab Shared Architecture is limited currently to running on the Microsoft Windows platform. A JAVA implementation of the Batched iLab Shared Architecture has been developed that can be used on other operating systems and still interoperate with the existing Microsoft .NET web services of MIT's iLab ServiceBroker. The Batched iLab Shared Architecture has been revised and separates the Labserver into a LabServer that handles experiment management and a LabEquipment that handles experiment execution. The JAVA implementation provides a 3-tier code development model that allows code to be reused and to develop only the code that is specific to each experiment.**

***Index Terms*—MIT iLab, Remote laboratories, Web Services.**

I. INTRODUCTION

The iLab Shared Architecture (ISA) developed by MIT [1] uses the Microsoft .NET (DotNet) web services of the Microsoft Windows platform [2]. It also uses the Microsoft SQL database server for information storage by the ServiceBroker and LabServers. The Microsoft Visual Studio development tools are used to build the web applications for the ServiceBroker, LabClients and LabServers. By developing these web applications in JAVA [3] and using the PostgreSQL database [4], it is now possible to extend the use of the iLab Shared Architecture beyond the Microsoft Windows platform.

JAVA provides the *jax-ws* framework for developing web service applications that interoperate with the DotNet web services. This allows a JAVA LabClient to communicate with a DotNet ServiceBroker that in turn communicates with a JAVA LabServer. The NetBeans IDE and Glassfish web server [5] are used to develop these JAVA web and web service applications while PostgreSQL is used for database storage.

JAVA is the programming language of choice with over 3 billion devices using JAVA. The development tools and database software are free to download from the Internet, are free to use, and are available for a wide range of operating system platforms such as LINUX and Mac-OS as well as Microsoft Windows.

II. ILAB SHARED ARCHITECTURE MODEL

A. *Existing Model*

Figure. 1 shows the existing model for an MIT Batched experiment which consists of three parts: a LabClient, a ServiceBroker and a LabServer with attached equipment.

The LabClient provides the interface through which the user creates and submits an experiment specification. The

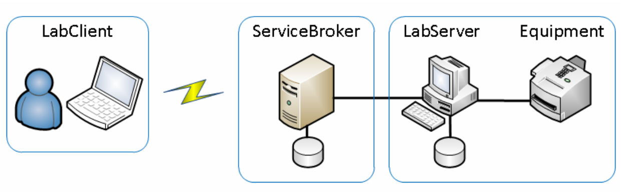


Figure 1. MIT's Batched iLab Shared Architecture model.

ServiceBroker enables the user to launch the LabClient after proper authentication. The LabServer handles the validation and submission of an experiment specification from the LabClient (via the ServiceBroker) and executes the experiment on the equipment according to the experiment specification.

B. *Revised Model*

Figure. 2 shows the model developed at the University of Queensland which consists of four parts: LabClient, ServiceBroker, LabServer and LabEquipment [6]. The LabClient and ServiceBroker are the same as in the MIT model but the LabServer has been separated into two parts. Again, the LabServer handles the validation and submission of an experiment specification from the LabClient (via the ServiceBroker) but experiment execution has been moved from the LabServer the LabEquipment.

Quite often the software used to drive the equipment hardware is very dependent on the computer platform being used, and in many cases is only available for the Microsoft Windows platform. So by separating out the LabEquipment from the LabServer, the LabServer can be developed in JAVA and handle experiment management while the LabEquipment can be platform dependent and handle experiment execution.

As a result of the separation, the LabEquipment and LabServer no longer need to reside on the same computer. The LabEquipment can reside at a location suitable for running the experiment which may be in a hazardous area or behind a network firewall. The LabServer can reside on a system server, possibly along with the ServiceBroker and the LabClient.

An example of this occurs at the University of Queensland where the Radioactivity LabEquipment is located in

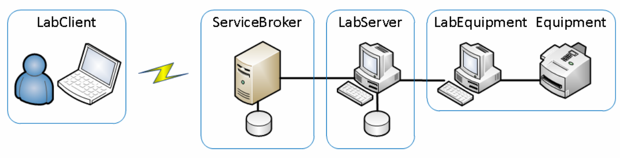


Figure 2. Revised Batched iLab Shared Architecture model.

the Physics department while the Radioactivity LabServer and LabClient reside on the School of Information Technology and Electrical Engineering server along with the UQ OpeniLabs ServiceBroker.

The LabEquipment also provides a mechanism for powering down the equipment after a period of inactivity. Generally, there is a burst of activity when experiments are submitted followed by long periods of inactivity. It makes sense then to power down the equipment during these periods of inactivity to reduce component wear as well as reducing overall power usage.

C. *LabEquipment Farm*

By having the LabEquipment separate from the LabServer, it is now possible to duplicate the LabEquipment to create a “farm” (Figure 3) of LabEquipment units connected to the same LabServer. This has the advantages of increased experiment throughput and improved reliability. Should any one of the LabEquipment units fail, the other units will pick up the load. The disadvantage is that the LabEquipment units must produce the same result, within acceptable limits, for the same experiment specification.

III. JAVA INTEROPERABILITY WITH DOTNET

A LabServer coded in JAVA must interoperate with a DotNet ServiceBroker in exactly the same way as a DotNet LabServer would (Figure 4).

JAVA provides a *jax-ws* framework to do this but this framework requires web service JAVA classes to function. Fortunately these classes can be generated from a WSDL file created from the DotNet web service.

The web service of a DotNet LabServer is opened with a web browser and then the *Service Description* (WSDL) is viewed in the browser. The service description is saved to a file which is then used to generate the web service JAVA classes.

IV. LABSERVER WEB SERVICE

The JAVA web service for the LabServer is generated from the WSDL file obtained from the MIT’s DotNet LabServer web service. An example implementation of a Batch LabServer was provided with the MIT 6.1 version of the Batch ServiceBroker as the *Time-Of-Day* experiment. This implementation was originally used to obtain the WSDL file from the DotNet web service. Since then, an abstract class of the DotNet web service has been written for the LabServer to obtain the WSDL file.

A. *SOAP Header*

The ServiceBroker passes information in the SOAP header of the web service call to the LabServer. This includes the *Identifier* which is the ServiceBroker's GUID and the outgoing *PassKey* and both are used to determine the authenticity of the ServiceBroker making the request. (Figure 5)

SOAP header processing is carried out in a message handler that is attached to the web service for the incoming requests. Since each request is independent of any other request, the information in the SOAP header has to be passed between the message handler and the web service application by means of the message context. The LabServer may receive two consecutive requests from two different ServiceBrokers meaning the information in the SOAP header will be different for each request.

The *Identifier* and *PassKey* are contained in an *AuthHeader* object that is extracted from the SOAP header and passed to the LabServer’s web service through the message context for authentication. Should authentication fail, an exception is thrown back to the ServiceBroker denying access to the LabServer.

The LabServer uses a JAVA Enterprise Bean to do the work of the web service. The web service simply processes the *AuthHeader* object information that it received through the message context from the message handler before passing the request on to the bean to do the work.

B. *Initialization*

The first point of contact with the web service is its message handler. When a ServiceBroker sends a request to the LabServer, the message handler processes the request before passing the message to the LabServer’s web service (Figure 6). This means that the first phase of initialization of the LabServer has to be carried out in the message handler and not in the web service. This is fine because a message context exists in the message handler allowing configuration information to be read from the *web.xml* file. This information includes the location of the configuration properties XML file which is read and an instance of a *ConfigProperties* object created.

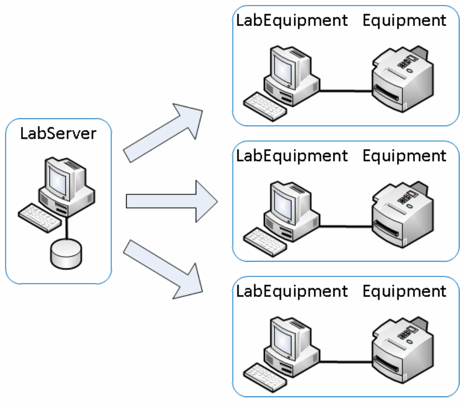


Figure 3. LabEquipment farm.

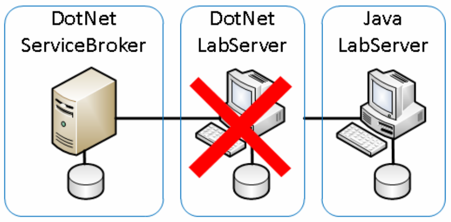


Figure 4. Java LabServer replaces DotNet LabServer.

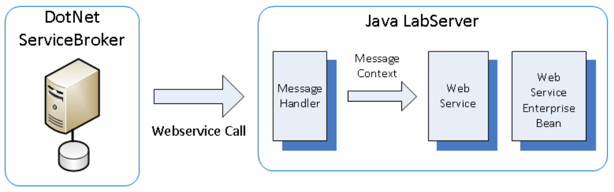


Figure 5. DotNet ServiceBroker web service call to a Java LabServer.

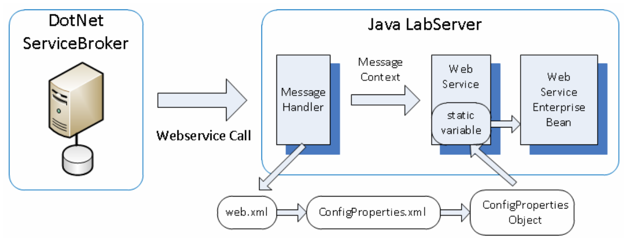


Figure 6. Java LabServer web service initialization.

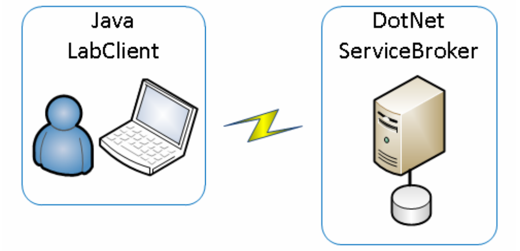


Figure 7. Java LabClient web application.

But how does the LabServer's web service Enterprise Bean get to see the configuration information? The message handler places the newly created *ConfigProperties* object into a static variable in the LabServer's web service and sets an *initialized* boolean flag. When the web service bean's constructor executes, it gets the *ConfigProperties* object from the static variable in the LabServer's web service and carries out the remainder of the initialization required by the LabServer.

Why can't the LabServer's web service bean get the configuration information from the *web.xml* file itself? A web service context does not exist outside of a web service call to enable that to occur.

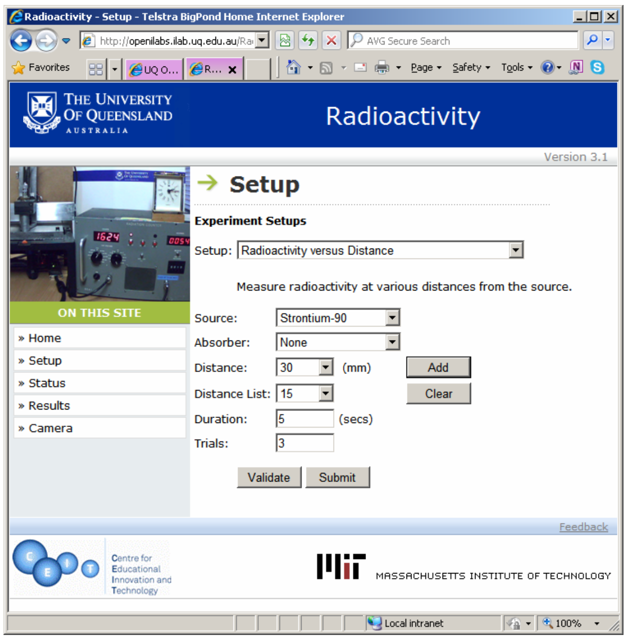
V. LABCLIENT WEB APPLICATION

The LabClient uses the *Java ServerFaces* framework to provide an interface for the user to submit experiments and retrieve results (Figure 7). A *Loader Script* is used by the ServiceBroker to launch the LabClient. The loader script passes the LabServer's GUID and the ServiceBroker's web service URL to the LabClient, by way of the URL request parameters. This allows a single deployment of the LabClient to be used by multiple ServiceBrokers and LabServers.

The JAVA web service reference for the LabClient is generated from the WSDL file obtained from the DotNet ServiceBroker web service abstract class that includes only the web service methods for batch experiments.

The ServiceBroker generates a *CouponId* and *CouponPasskey* when the LabClient is launched and passes these to the LabClient also by the way of the URL request parameters. The *CouponId* and *CouponPasskey* are then passed back to the ServiceBroker in the SOAP header with each web service call and used by the ServiceBroker to determine the authenticity of the LabClient making the request.

The LabClients developed at the University of Queensland are considered to be an engineering approach. They do not provide any fancy graphical interface but do provide sufficient information to submit an experiment to the LabServer. For example, the LabClient for the Radioactivity experiment simply provides standard web page controls to specify the experiment setup, distance of the Geiger tube


Radioactivity - Setup - Telstra BigPond Home Internet Explorer
http://openilabs.ilab.uq.edu.au/Ra
AVG Secure Search
Favorites
Page
Safety
Tools
THE UNIVERSITY OF QUEENSLAND AUSTRALIA
ON THIS SITE
» Home
» Setup
» Status
» Results
» Camera
Add
Clear
Validate
Submit
Feedback
Centre for Educational Innovation and Technology
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Local intranet
100%

Radioactivity

Version 3.1

Setup

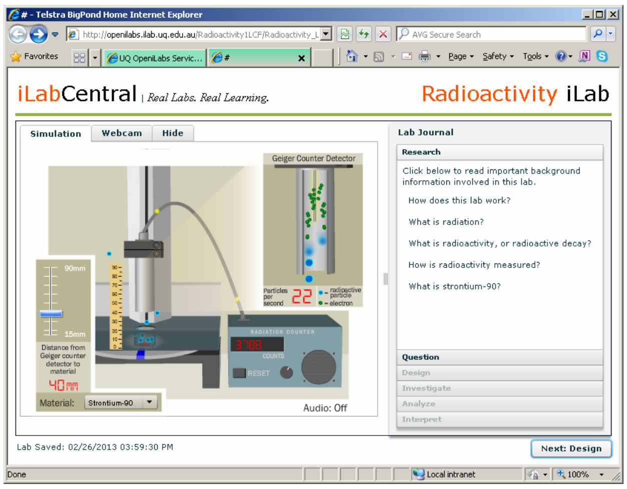
Experiment Setups

Setup: Radioactivity versus Distance

Measure radioactivity at various distances from the source.

Source: Strontium-90
Absorber: None
Distance: 30 (mm)
Distance List: 15
Duration: 5 (secs)
Trials: 3

Figure 8. University of Queensland's Radioactivity LabClient.



Lab Journal

Research

Click below to read important background information involved in this lab.

How does this lab work?

What is radiation?

What is radioactivity, or radioactive decay?

How is radioactivity measured?

What is strontium-90?

Question

Design

Investigate

Analyze

Interpret

Lab Saved: 02/26/2013 03:59:30 PM

Figure 9. NorthWestern University's Radioactivity LabClient.

from the radioactive source and the duration of exposure to the radioactive source as shown in Figure 8.

Northwestern University has developed an Adobe Flash LabClient [7] for use by high school students as shown in Figure 9. It provides a graphical simulation of the Radioactivity experiment and then a step-by-step procedure for preparing, running and completing the experiment. The students are asked questions and are required to provide answers to those questions before continuing to the next step.

A. SOAP Header

The LabClient passes information in the SOAP header of the web service call to the ServiceBroker. This includes the *CouponId* and *CouponPasskey* which were passed by the ServiceBroker to the LabClient when it was launched.

SOAP header processing is carried out in the message handler that is attached to the web service reference for the outgoing requests. The *CouponId* and *CouponPasskey* are passed to the message handler by the LabClient through the message context where they are placed in an *SbAuthHeader* object and inserted into the SOAP header.

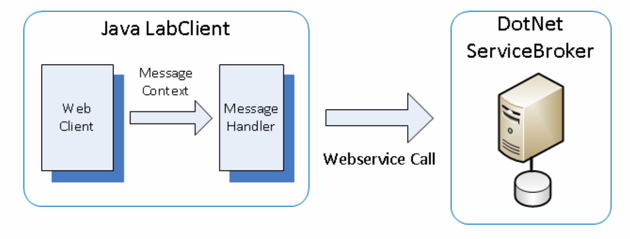


Figure 10. Java LabClient web service call to a DotNet ServiceBroker.

VI. LABEQUIPMENT WEB SERVICE

The LabEquipment web service application is responsible for executing an experiment on the equipment according to the specification provided to it by the LabServer. In certain cases, it also provides a mechanism for powering down the equipment after a period of inactivity. Generally, there is a burst of activity when experiments are submitted followed by long periods of inactivity. It makes sense then to power down the equipment during these periods of inactivity to reduce component wear as well as reducing overall power usage.

As mentioned earlier, the LabEquipment software is dependent on the computer platform used. If the LabEquipment only used the network to communicate with the equipment or carry out simulations then the LabEquipment could be developed in JAVA.

The JAVA web service reference for the LabEquipment is generated from the WSDL file obtained from the DotNet LabEquipment web service abstract class.

A. *SOAP Header*

In a similar fashion to the LabServer web service, the LabServer passes information in the SOAP header of the web service call to the LabEquipment. This includes the *Identifier* which is the LabServer's GUID and the outgoing *PassKey* and both are used to determine the authenticity of the LabServer making the request.

SOAP header processing is carried out in the message handler that is attached to the web service for incoming requests. The *AuthHeader* object containing the *Identifier* and *PassKey* is extracted from the SOAP header and passed to the LabEquipment's web service through the message context for authentication. Should authentication fail, an exception is thrown back to the LabServer denying access to the LabEquipment.

The LabEquipment uses a JAVA Enterprise Bean to do the work of the web service. The web service simply processes the *AuthHeader* object information that it received through the message context before passing the request on to the bean to do the work (Figure 11).

B. *Initialization*

Initialization of the LabEquipment web service occurs in the same way as the LabServer web service. The first phase of the initialization occurs in the message handler attached to the web service. The web service bean's constructor then carries out the remainder of the initialization required by the LabEquipment.

C. *Web Methods*

The LabEquipment service provides a number of web methods that can be called by the LabServer to run experiments on the LabEquipment. These include:

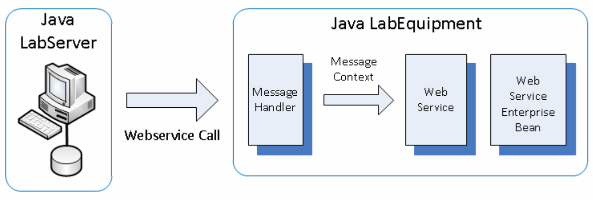


Figure 11. Java LabServer web service call to a Java LabEquipment.

- *GetLabEquipmentStatus* – Determines the status of the LabEquipment and if it is offline, provides a status message.
- *Validate* – Takes an experiment specification string in XML format and determines the validity of the specification parameters and the estimated execution time. This time is dependent on the experiment specification and the type of equipment used.
- *StartLabExecution* - Takes an experiment specification string in XML format and after successful validation, starts the experiment executing on the equipment.
- *GetLabExecutionStatus* – Determines the status of the currently executing experiment including an estimate of the execution time remaining.
- *GetLabExecutionResults* – Retrieves the results of the experiment as a string in XML format after it has finished executing.
- *CancelLabExecution* – Cancels the experiment that is currently executing.

These web methods do not depend on the experiment that is being executed or the type of equipment being used.

D. *Experiment Execution*

Execution of the experiment is carried out by the LabEquipment under the management of the LabServer and is summarized by the flowchart shown in Figure 12.

The LabServer starts the experiment executing on the LabEquipment by calling the *StartLabExecution* web service method. Periodically, the LabServer checks the execution status of the experiment by calling the *GetLabExecutionStatus* web service method. This provides the LabServer with an estimate of the time remaining until completion and allows the LabServer to determine when to check the execution status again provided that execu-

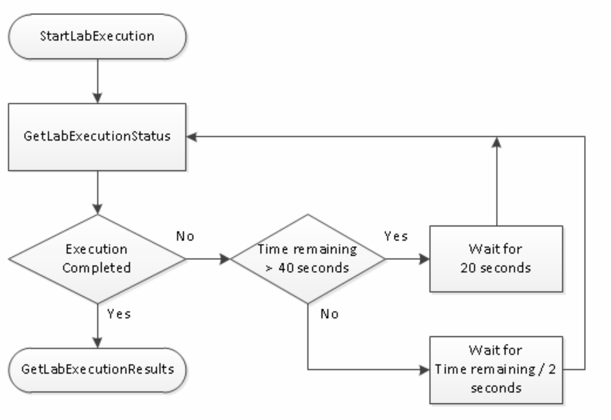


Figure 12. LabServer experiment execution management flowchart.

tion has not completed, failed or been cancelled. The current implementation of the LabServer checks the execution status every 20 seconds which prevents excessive traffic between the LabServer and LabEquipment. As the time remaining until completion approaches zero, the LabServer checks the execution status more often by halving the time remaining.

When the experiment execution completes on the LabEquipment, the LabServer retrieves the results by calling the *GetLabExecutionResults* web service method.

VII. DUMMY SERVICEBROKER

A Dummy ServiceBroker has been developed to enable the development of the LabServer and its LabClient without the complexities of having to log into an iLab ServiceBroker. The Dummy ServiceBroker simply provides pass-through methods to allow the LabClient to communicate directly with the LabServer. Only one web method is not entirely pass-through and that is the *Submit* web method where an experiment number needs to be generated.

It is then possible, while debugging, to step through the code from the LabClient into the Dummy ServiceBroker then into the LabServer and LabEquipment and all the way back again to the LabClient.

The Dummy ServiceBroker can also communicate with more than one LabServer during development. This may be useful when one LabServer is being developed with the JAVA *jax-ws* framework and while another LabServer is being developed with the Microsoft .NET framework.

VIII. 3-TIER CODE DEVELOPMENT MODEL

Development of the LabServer and LabEquipment web service applications occurs at three levels as shown in Figure 13.

For LabServer applications, the bottom level is a library containing code common to all LabServer web service applications. It contains the base classes for processing experiment specifications and experiment results, database routines and the process threads that manage experiment execution on the LabEquipment.

For LabEquipment applications, the bottom level is a library containing code common to all LabEquipment web service applications. It contains the base classes for processing experiment specifications and experiment results, base classes for the equipment devices and experiment execution drivers and the process threads that power up and power down the equipment.

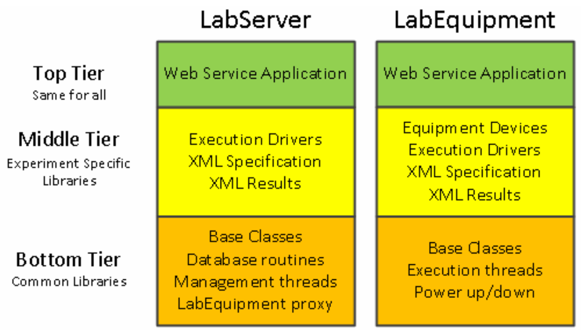


Figure 13. 3-Tier code development model.

The next level up is the library containing the code that processes the experiment specification and experiment results for a specific experiment, for example, the Radioactivity experiment or the Time-Of-Day experiment. For the LabServer, this level also contains the drivers that manage experiment execution for specific experiments. For the LabEquipment, this level also contains the equipment devices and experiment execution drivers specific to the experiment.

The top level of the model is the web service application and its message handler. The code at this level cannot be placed in a library because it is the application that is deployed to the web server. The web service applications for each LabServer are identical except for configuration information. Similarly, the web service applications for each LabEquipment are identical except for configuration information.

Using this model for the LabServer and LabEquipment allows speedy creation of new applications by focusing on the development of experiment specific code at the second level and reusing the code at the other two levels.

IX. CONCLUSION

The development of a JAVA implementation of the Batched iLab Shared Architecture has enabled platforms other than Microsoft Windows to host iLab experiments.

The use of the JAVA *jax-ws* framework has allowed the LabServer web service applications and LabClient web applications to interoperate with existing Microsoft .NET iLab ServiceBrokers.

By using the 3-tier code development approach, the time and effort required to create new iLab experiments is reduced.

REFERENCES

[1] J. Harward, et. al., "The iLab Shared Architecture: A Web Services Infrastructure to Build Communities of Internet Accessible Laboratories", *Proceedings of the IEEE*, Vol96(6), pp. 931-950, June 2008. http://dx.doi.org/10.1109/JPROC.2008.921607
[2] *iLab Downloads – iLabs Dev – MIT Wiki Service* https://wikis.mit.edu/confluence/display/ILAB2/iLab+Downloads
[3] *Java Platform (JDK) 7u9,* http://www.oracle.com/technetwork/java/javase/downloads/
[4] *PostgreSQL 9.2,* http://www.postgresql.org/download/
[5] *NetBeans IDE 7.2 + Glassfish Development Server 3.1.2,* http://netbeans.org/downloads/
[6] *UQ-iLab-BatchLabServer-Java Repository,* https://github.com/uqlpayne/UQ-iLab-BatchLabServer-Java
[7] ilabCentral – The place to share remote online laboratories http://ilabcentral.org

AUTHORS

L. J. Payne is with the School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia (e-mail: uqlpayne@uq.edu.au)

M. F. Schulz is with the Centre for Educational Innovation & Technology, The University of Queensland, Brisbane, Australia (e-mail: m.schulz@uq.edu.au).

This article is an extended and modified version of a paper presented at the International Conference on Remote Engineering & Virtual Instrumentation (REV2012), held at University of Deusto, Bilbao, Spain, July 4-6, 2012. Received 27 February 2013. Published as resubmitted by the authors 20 March 2013.