

Controlling Remote Instruments Using Web Services for Online Experiment Systems

Yuhong Yan¹, Yong Liang², Xinge Du²

¹ NRC-IIT, Fredericton, NB, Canada,
Yuhong.yan@nrc.gc.ca

² Faculty of Computer Science, UNB, Canada
{Yong.liang, Xinge.Du}@unb.ca

Abstract

Online experimentation allows students from anywhere to operate remote instruments at any time. This promising e-learning application is well positioned to use Web Services to conduct online experiment systems due to its interoperability and Internet compliance. We present a double client-server architecture for online experiment systems and the methodology to wrap the functions of instruments into Web Services. We propose that the instrument Web Services should be stateful services and we present the framework to manage the states of the instrument web services. We benchmark the performance of this system when using SOAP as the wire format for communication and propose solutions to optimize performance.

1. Introduction

In the scope of e-learning, the goal of online experiment systems is to provide the ability for students to conduct experiments via the Internet. Online experiment systems can provide easy access to experiments for educational institutions that cannot afford the experimental equipment, and can increase the effectiveness of online learning. Online experiment systems are studied in several European projects, such as Emerge [5] and Prolearn [13], which involve about nine universities across Europe. MIT's weblab [7] is among several prototypes that provide students access to remote instruments via web interface. In Canada, the CFI-funded online lab at Tele-university offers similar functions [15].

Based on the functions of the online laboratories, online experiment systems can be classified as *virtual laboratories* that provide a simulation environment of the experiments, and *remote laboratories* that allow students to operate the remote instruments via a graphical user interface (GUI). The online experiment system for the

remote laboratories is studied in this paper. The technology adapted by the current online experiment system is based on simple client-server architecture and uses off-the-shelf middleware for communication. It is difficult to connect heterogeneous resources for experiments using off-the-shelf middleware. Normally, an online system relies on products from individual companies, such as National Instruments or Agilent. WindowsTM is the common operating system for these instruments. The client side has to install proper software to operate the remote instruments. The goals of resource sharing among the online laboratories and easy access via the web remain unachieved.

Web Services, as the latest technology for distributed applications, provides a new potential to build online experiment systems. The most valuable feature of Web Services for online experiment systems is interoperability. By sending eXtensible Markup Language (XML) based Simple Object Access Protocol (SOAP) [19] message to the remote components, and using Internet protocols, such as Hypertext Transfer Protocol (HTTP) or Blocks Extensible Exchange Protocol (BEEP), Web Services ensures the interoperability of the components on different platforms and/or implemented in different programming languages. Other Web Services standards can also play a role in this application. For example, Universal Description Discovery Integration (UDDI) [17] can serve to describe the scattered experimental resources, and Business Process Execution Language (BPEL) [2] can be used to organize learning procedures.

Although Web Services has strong advantages on interoperability, it has intrinsic weaknesses on latency and scalability because it uses more transport layers. For online experimentation, this would cause problems because of the need to transport large volume of data between the services and the clients. Classic Web Services are stateless. This also needs to be examined in this application. Fundamentally, we need to investigate

the methodology to wrap the instrument functions as Web Services. In this paper, we present our results in using web services for Online Experiment Systems (OES). We propose a service-oriented architecture for OES and present the methodology to wrap the functions of instruments into Web Services. We discuss the requirements of stateful services and the performance issues in this application caused by using SOAP. Though some of the discussions in this paper can be applied to the e-science domain, we would like to limit the scope of our applications to e-learning, where the experiments are not mission-critical and the instruments are standard commercial products with standard Application Program Interfaces (API).

This paper is organized as follows: Section 2 presents the general framework; Section 3 presents the methodology to wrap the instrument functions as Web Services; Section 4 discusses the management of stateful instrumental Web Services; Section 5 benchmarks the performance of Web Services in this application and presents the optimization methods to improve performance; and Section 6 is the conclusion.

2. The Web Service-based Framework for Online Experiment System

An *Online Experiment System (OES)* uses the scattered computational resources and instruments on the networks for experiments. The online laboratory system we present here is a *web-enabled distributed system*. It has two-fold meanings: the user accesses the online experiment system via the web interface; and the heterogeneous resources and devices interoperate with each other using Web Services protocols. The goals of this framework are: 1) to share the experimental resources among different labs via the Internet; 2) to increase the ability of computation and data-sharing among different labs, and; 3) to enable users to access online labs at any time and from anywhere either for self-learning or for collaborative laboratory work sessions.

Figure 1 shows the *double client-server architecture* for an online experiment system. The first client-server architecture is between the client browser and the web server associated with the online lab management system. The user connects to the online laboratory using a web browser. The web server is used to render the GUI interface (see the next section) on the web browser. The second client-server architecture is between the online lab management system and the scattered resources that are wrapped as Web Services. The remote services have their own web server to receive the SOAP requests from the online lab management system. Web Services are used for communication between the online laboratory and the remote resources.

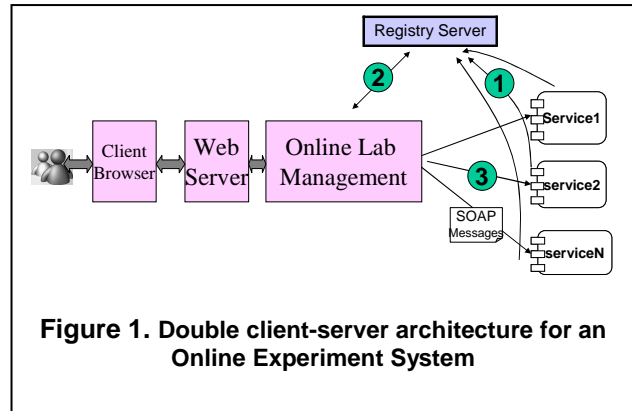


Figure 1. Double client-server architecture for an Online Experiment System

The system works in a series of steps. A service provider first registers its services in a UDDI registry server (step 1 in Figure 1). A service requester searches the registry server and gets all the potential resources. It selects the proper services based on its own criteria (step 2). The service requester sends SOAP messages directly to the service provider to invoke the remote service (step 3).

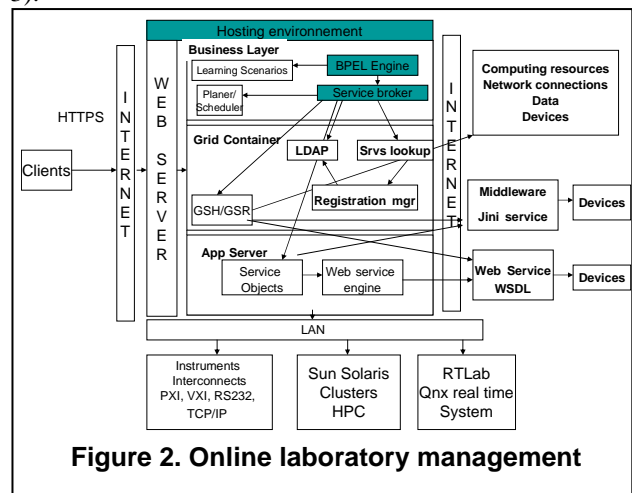


Figure 2. Online laboratory management

Figure 2 shows the internal structural of the online laboratory management system. It needs to manage the local resources on the local area network (LAN) and the remote resources connected by the Internet. Among the resources to be managed are the computational resources which can be managed using Grid Service techniques. The rest of the resources are experimental instruments which can be managed by Web Services as described in the next section. As such, the back-end uses both Grid Services and Web Services. It uses a web server for the front-end representation, and has three layers in the back end. The top layer is the logic layer where the learning scenarios are defined and the processes are managed. The learning scenarios are defined in four aspects: learning objects, a pedagogical model, a media model and distribution [10]. Among those, the pedagogical model defines the process of a course. The process is translated

directly into Business Process Execution Language (BPEL). The *BEPL engine* is a tool to monitor and control the process automatically. The BEPL engine is able to automatically invoke remote Web Services. The activities in a learning scenario may need remote web services. The *Service Broker* determines if the services come from local services (e.g. the blocks under the “LAN”), or remote external services (e.g. the blocks of “jini services”, “web services”). The Service Broker knows the various protocols used for remote services. For Grid services, it sends the requests to the Grid Service Handler/Grid Service References (GSH/GSR) in the *Grid Container*. The GSH/GSR is a mechanism in Grid Service to get the reference of the remote objects and forward the requests to the remote objects. GSH/GSR is able to invoke the services either in middleware, (e.g. jini), or in Web Services. Service Broker can also invoke Web Services without the GSH/GSR interface by sending the request to the service objects in the application server (the bottom layer). Service Broker regularly calls the *Service Lookup* (“*srv lookup*” in Figure 2) and updates the local *LDAP* with the results. *Registration Manager* (“*Registration Mgr*” in Figure 2) helps to convert information from a service registry into LDAP. The bottom layer is the *Application Server* layer. The Application Server provides flexible mechanisms to manage the *Service Objects* and interface to the *Web Service Engine*. Service Objects are some software components that process the data from remote web services. See the next section for an example of service implementation. The Web Service Engine sends the SOAP message to invoke the remote web services. This framework works with the computing resources using Grid protocols, software components using middleware, and Web Services components. We believe it covers all the resources needed for online experiments. If Grid Services will merge with Web Services in the future, maybe the two lower layers in Figure 2 will, at some point, be united into one layer. But in our current study, we find we still need to use different techniques to manage different resources.

In the following two sections, we discuss how to use Web Services to control the instruments. Due to the nature of the instruments, we need to custom design the Web Services especially for this application. Grid Services is not suitable for controlling instruments due to its inflexibility and bounding to computational resources.

3. Wrapping Instrument Operations as Web Services

A Web Service Description Language (WSDL) [18] file contains the operations of the web service and the arguments to invoke operations. When instrument functions are wrapped as Web Services, the interface of

the instrument web service is described in a WSDL file. An instrument service needs to provide three kinds of information: 1) the input/output parameters to operate the instrument; 2) the information about rendering the GUI of the instrument panels; and, 3) the metadata about the instruments. These issues are described individually below.

3.1 Generic Approach to Wrap Instrument Operations based on VISA standard

Instrument I/O is a well studied domain for which industrial standards have been established. Two methods to control instruments are by using an instrument driver or by making direct calls to the I/O library. If using an instrument driver, the user will call functions that cause the instrument to take some action. If using the I/O library, the user will control the instrument by sending an ASCII string to it and reading ASCII strings back from it. The commonly used languages to operate instruments are C, C# or Visual Basic. The commonly accepted industrial standards are Virtual Instrument System Architecture (VISA) and Interchangeable Virtual Instruments (IVI) [1]. Most commercial products follow these standards. The purpose of these standards is to enable interoperability of instruments, which means using common APIs of the instruments. Therefore, it is possible to generate generic WSDL interfaces for instruments based on these standards. The relationship between VISA and IVI is shown in Figure 3. The individual instruments – Instr. A, B and C – have their own drivers. These drivers are wrapped by VISA compliant drivers. The IVI compliant drivers are built still on the top of VISA standard.

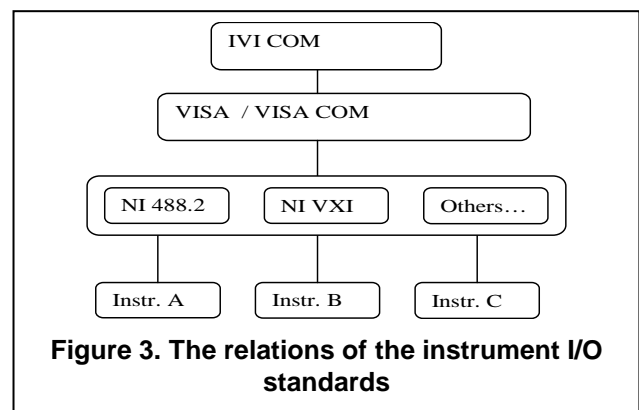


Figure 3. The relations of the instrument I/O standards

Both VISA and IVI standards operate the instruments by reading and sending ASCII strings to the instruments. Compared with VISA, IVI can operate the instrument by referencing its properties. The IVI standard classifies the instruments into eight classes. Each class has basic properties that are shared by all the instruments in the same class, and extension properties that are unique to the

individual instrument. As an example, Table 1 shows the code to set the frequency of an Agilent Waveform Generator 33220A to 2500.0HZ, using IVI COM. Table 2 is the code of VISA COM to implement the same function. Using VISA COM, people do not know the semantics of the parameters. That is to say, setting the Frequency or Voltage, people will use the same API.

Table 1. Sample code of IVI COM

```
IAgilent33220Ptr Fgen;
.....
Fgen->Output->Frequency = 2500.0;
.....
```

Table 2. Sample code of VISA COM

```
Fgen->WriteString("FREQuency 2500")
```

We consider that using the VISA standard, the methodology of wrapping the instrument services can be generic to any of the instruments, which means that many instruments can share the same Web Services interface. Indeed, using the VISA standard, we need only to define an operation *writeString* for sending commands or data to the instrument. The argument of this operation is always string, which is the same for any instrument. Table 3 is the snippet of WSDL for defining the operation of *writeString*. Similarly, we can define an operation *readString* for getting status or data from the instrument, which is eliminated from Table 3.

Table 3. The snippet of WSDL to operate an instrument

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions .....>
.....
  <!--define the response message -->
  <wsdl:message name="writeStringResponse">
    <wsdl:part name="writeStringReturn"
      type="xsd:int"/>
  </wsdl:message>
  <!--define the request message -->
  <wsdl:message name="writeStringRequest">
    <wsdl:part name="in0" type="xsd:string"/>
  </wsdl:message>
  <!--define the operation -->
  <wsdl:operation name="writeString"
    parameterOrder="in0">
    <wsdl:input message="intf:writeStringRequest"
      name="writeStringRequest"/>
    <wsdl:output
      message="intf:writeStringResponse"
      name="writeStringResponse"/>
  </wsdl:operation>
.....
</wsdl:definitions>
```

In the example in Table 4, we demonstrate how to operate the waveform generator to generate a sinusoid waveform. The set of control parameters for the sinusoid waveform contains “*instrument address*”, “*wave shape*”, “*impedance*”, “*frequency*”, “*amplitude*”, and “*offset*”. In order to improve performance by reducing the time taken to send SOAP messages (ref Section 5), those parameters are put into one string. This means that only one SOAP message is transported to pass all the parameters from the client to the server. After the server gets the string from the client, it will parse the string according to the delimiter (here we use “|”) and send the command to the instrument.

Table 4. Sinusoid waveform parameters in one string

```
"*RST|FUNction SINusoid|OUTPut:LOAD
50|FREQuency 2500|VOLTage
1.2|VOLTage:OFFSet 0.4|OUTPut ON";
```

Although we prefer to use the VISA standard to wrap the instrument functions, it is also possible to use the IVI standard. The difference is that each instrument class will have a common WSDL file in which the operations for the basic properties of this instrument class are defined. For instruments having extension properties, the WSDL has to be generated separately to include the operations for the extension properties. Therefore, if using the IVI standard, the interoperability is satisfied if the instruments are in the same class and if they have the same extension properties.

3.2 Design the Web GUI for the Instrument

The panel of a remote instrument should be displayed graphically on a web browser. The user operates the GUI to control the instruments. The methodology to describe instrument panels is presented in [6]. The principle is to design an XML schema which defines the syntax of the panel of a kind of instruments. An XML file compliant to the schema describes the panel of an individual instrument. Then the XML file can be parsed and rendered at the client side. We use the multimeter Agilent 34401A as an example. A snippet of the XML for its panel [20] is in Table 5.

One can see the container panel objects are the *parentFrame*, *parentPanel* and *childPanel*. A container object can contain other panel objects, such as labels and text boxes. A container object has a layout that describes how to render the objects inside the container. If one is familiar with java, one can see the objects can be mapped one by one to the classes in a java swing GUI package.

Table 5. A snippet of the XML to describe the panel of Agilent 34401A

```

<parentFrame parentFrameName="Frame
Container">
  <parentFrameLayout> ... </parentFrameLayout>
</parentFrame>
<parentPanel parentPanelName="Parent Panel">
  <parentPanelLayout>GridBagLayout</parentPanel
Layout>
<parentPanelDimension>...</parentPanelDimension>
</parentPanel>
<childPanel childPanelName =
"ExternalParametersChildPanel">
  <childPanelLayout> ... </childPanelLayout>
  <component className="JLabel">
    <componentName> ... </componentName>
    ...
  </component>
  ...
</childPanel>

```

Figure 4 shows the principle to display the panel from its XML description. The XML schema for the Digital Multimeter is in DMM_GUI.xml. It validates the file DMM_Agilent_34401A_GUI.xml which defines the GUI for the Agilent 34401A. The JAXB is used to parse DMM_Agilent_34401A_GUI.xml. Then a java servlet is used to display the panel object on an HTML page. The generated GUI page is displayed on the right bottom section of Figure 4.

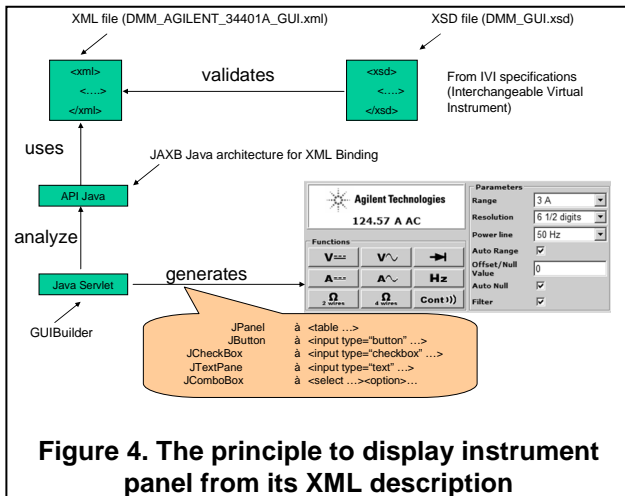


Figure 4. The principle to display instrument panel from its XML description

In the double client-server architecture, the XML files are at the instrument service site, and can be downloaded to the online laboratory management system (see the previous section). The java servlet for rendering the GUI resides in the web server for the online laboratory management system. The client of the end user only requires a normal web page. Therefore, we have a thin client. If we need to show arbitrary shapes, such as waveforms, it is a little more complex. There are two options. If we want to achieve zero installation at the

client side, i.e. no code to be installed, we can generate a jpg image for the waveform. This is a mature technology. If we allow the client side to use applets (for java) or activeX control (for windows platform), we have a thicker client. This requires simple coding.

3.3 Interfaces of Meta Information.

The IEEE Learning Object Metadata (LOM) standard defines metadata for a learning object [8]. LOM is designed for the objects of an online course. It includes information such as the author, the organization, and the language. Though we can view an experiment as a course, which also has the tutorial material and assignment, we need other special information to describe the status of an instrument. In [3], the LOM standard is extended for experimentation context. For operating an instrument, two additional types of information, the *availability* and the *quality of services (QoS)*, are required.

The LOM information is defined in an XML file. In the WSDL, we define the operation, *getLOMMetaData*, to download the information.

The availability is important for booking the service. We have an operation, *getAvailabilityInfo*, to get this information. The user can list all the available timeslots during a time interval, or query if the instrument is available for a specific time period.

Table 6. The operations to get metadata information in WSDL

```

<!--define the operation -->
<wsdl:operation name="getLOMMateData">
  <wsdl:output name="getLOMDataResponse">
  </wsdl:output>
<!--define the operation -->
<wsdl:operation name="getAvailabilityInfo">
  <wsdl:output name="getAvailabilityResponse">
  </wsdl:output>
<!--define the operation -->
<wsdl:operation name="getQoSInfo">
  <wsdl:output name="getQoSResponse">
  </wsdl:output>
</wsdl:operation>

```

QoS information is accumulated from history and can become an important selling and differentiating point of Web Services with similar functionality. We record the successful connecting rate to the instrument, the response time to the instrument, and customer's rating to use its service. QoS information is used when selecting available instruments for an experiment. The higher QoS of the instrument service, the more likely the OES selects this instrument or recommends it to the user to use. The operation *getQoSInfo*, is designed for this. Table 6 displays these operations in WSDL. These operations are

out typed operations (i.e. only have response SOAP message).

4. Managing Stateful Instrument Web Services

Instrument Web Services involve remotely operating real devices in real time. Improper design of the Web Services can cause damage to the instrument, and can lead to false measurement and control, which in turn will result in failure of the online experiment. In [20], we present the special requirements for the instrument Web Services, such as reliability mechanisms and communication strategies. By using proper software technologies, these requirements can be satisfied. In the following subsections, we will focus on how to manage the instruments as resources.

4.1 Stateful Service for Stateless Resources.

It is well known that classic Web Services is stateless, i.e. it does not maintain states between different clients or different invocations. HTTP, the commonly used transport protocol for Web Services, is a stateless data-forwarding mechanism. There are no guarantees of packets being delivered to the destination and no guarantee of the order of the arriving packets. Classic Web Services are suitable for services providing non-dynamic information. In this subsection, we discuss if additional effort is needed to manage the instrument Web Services.

An instrument itself is a stateless resource. This is because an instrument itself does not record client information or invocations. Indeed, an instrument acts in a reactive way. It receives commands, executes them accordingly, and returns the results. If we say an instrument has “states”, these are the parameters of its working mode, which have nothing to do with the states of a web service.

An instrument can only be occupied by one user at a time. Unlike the resources in Grid Services, instruments can only accept one user at a time because an instrument needs to be set to a specific working mode before it can work for a certain experiment. Normally it is not possible to recover an instrument’s status without a proper procedure, so many mechanisms in Grid Services are not useful in our application. The use of an instrument is booked by time slots. On some occasions, the tasks of an instrument can be managed by a queue [7].

An instrument normally does not need reliable communication. For real time control, only the current status matters. Past observations are not relevant. So instead of using the reliable communication mechanism to adjust the loss, errors or congestion, it is often better to send the latest data instead of re-sending the old data [16].

An instrument service needs to be stateful for two reasons. It must be stateful when it needs to record the operations from one user for payment accounting or to control how the user can use this instrument, and also when the results need to be transported among several resources asynchronously. In the next subsection, we present a method to build the stateful instrument web services.

4.2 Design the Stateful Service for Instrument Resources

As stated previously, we know that the instrument service has to identify clients and maintain a history of the operation. This kind of stateful service is different from the available stateful framework in Grid Services and WSRF. We design the stateful service for instrument resources as in Figure 5. The states are managed by the resource management layer. The client ID is transferred in SOAP to identify the states of the services. In detail:

- (1) The client sends the request to the web service. The request should contain the ID of the client to identify the session.
- (2) The web service returns the identifier of the reference.
- (3) The client always contacts the service using the resource identifier.
- (4) – (5) The online experiment is executed and the results are returned to the Web Service.
- (6) The Web Service records the results in a proper manner and returns the results to the client.

Compared to Grid Services, no service factory is needed, because an instrument service is a single user service, thus no service instances are created. Compared to WSRF, the resource itself remains stateless because it can be changed. The web service adds a layer to manage the states. The state management can be implemented by using a database.

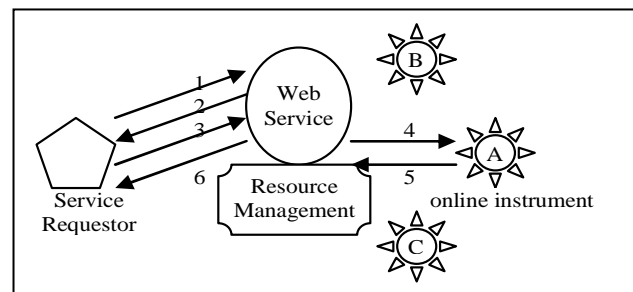


Figure 5. The stateful service for instrument resources

5. The performance Issues for Web Services for Online Experiment

The trade-off of the high interoperability of Web Services is its lower performance. Web Services have intrinsic performance weaknesses for two main reasons: *there are more transport layers than for middleware*; and *the overhead of using SOAP*. Many researchers have analyzed the problem of SOAP efficiency and identified some factors that can affect the latency performance of Web Services and SOAP [9][10][14]. For each factor that could cause the latency, there are some proposed methods to improve the performance. In this paper, we benchmark the SOAP efficiency in this context and propose the solutions to improve performance.

5.1 Benchmark of Latency

This benchmark test is aimed at determining the time to transport a service request from the requester to the provider. The time involves marshalling the SOAP message and binding it to the HTTP protocol at the request side, and the transportation time and decoding time on the service side. This test takes place when the instrument web service and the OES are on the same host, thus, the delay by the Internet is not considered. In Section 3, we described that instruments accept ASCII strings as input according to VISA and IVI standards. Therefore we use ASCII strings for encoding a volume of the floating numbers in SOAP message. In our test, we assumed each of the floating numbers had 16 digits to provide adequate precision. Therefore the size of the strings for floating numbers is directly proportional to the number of digits. We measured the time delay starting before the call of the service and ending as the request reaches the service endpoint.

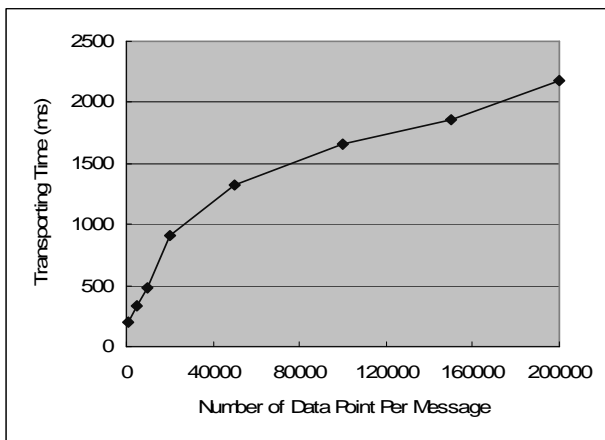


Figure 6. The delay vs. number of data point

Figure 6 shows the relation of the delay time vs. the number of data points per message. One can see that the delay increases quasi-linearly as the data points increase.

There is also a basic overhead for the transportation, which is primarily the time for setting up the TCP/IP connection.

5.2 Optimize the SOAP Efficiency

Latency of SOAP message is caused by the time of transportation, which is proportional to the size of SOAP, and the delay caused by the TCP/IP layer.

The most straightforward method of optimization is to reduce the SOAP message size by extracting the string out of the XML, compressing it into binary format (we use ZIP compression format here) and sending it as an attachment. The size of the payload is reduced to approximately 40 to 50 per cent of its original size. The SOAP messaging protocol supports Multipurpose Internet Mail Extensions (MIME) or Direct Internet Message Encapsulation (DIME) attachments. The difference is that MIME is designed to provide flexibility, while DIME is designed to be simpler and to provide more efficient message encapsulation. The results of applying different attachment approaches are shown in Figure 7. One can see that the transportation time can be reduced dramatically by compressing the SOAP content.

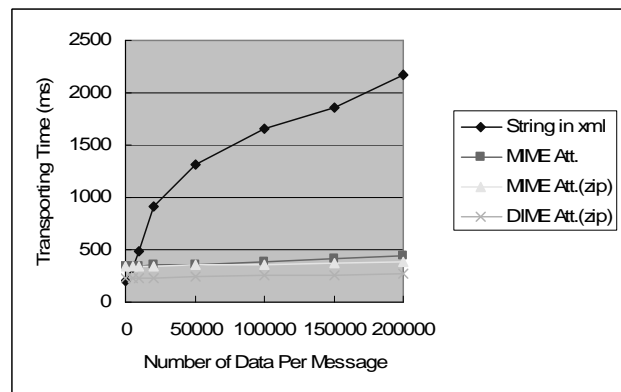


Figure 7. Different methods to send string data through SOAP

We can also optimize the underlying HTTP and TCP protocols for SOAP messaging. We present the possible methods below without testing results:

Persistent HTTP Connection. Persistent connection could “keep-alive” a connection and save the time needed to establish HTTP connection every time. For HTTP 1.0, the persistent connection works only if there is no proxy between the client and server. For HTTP1.1, the persistent connection can be used with more than one proxy between a client and a server.

Disable Nagle Algorithm and Remove TCP Delay ACK. The Nagle algorithm in combination with the TCP delayed ACK (the acknowledge response in TCP) are used to prevent network congestion [10], but they cause

unnecessary delays when sending a SOAP message [4]. It is possible to disable Nagle on both the server and client side to get considerable improvement for the response time.

Better Pipelined Connection by Using HTTP 1.1.

The use of HTTP inherits some of the TCP features such as the three-way handshake. This can cause delays. HTTP1.1 attempts to solve these problems. The result shows that HTTP1.1 can reduce the RTT (Round Trip Time) to half of HTTP1.0 implementation [11].

6. Conclusions

In this paper, we propose to wrap the remote instruments as Web Services for online experiment systems. The advantage of Web Services is its interoperability across platforms and programming languages. Its trade-off is low efficiency caused by SOAP messaging. This paper covers the essential issues to build such instrument Web Services, such as WSDL design, stateful service management and performance issues. The future work would be to continue optimizing the SOAP messaging and to analyse the resource description and integration issues.

References

- [1] Agilent Inc. About Instrument I/O http://adn.tm.agilent.com/index.cgi?CONTENT_ID=239, 2005.
- [2] Andrews, T., F. Curbera, et al., (2004), Specification: Business Process Execution Language for Web Services Version 1.1, <http://www-128.ibm.com/developerworks/library/ws-bpel/>
- [3] Bagnasco, A., M. Chirico, A. M. Scapolla, (2002) XML Technologies to Design Didactical Distributed Measurement Laboratories, IEEE IMTC2002, Anchorage, Alaska, USA.
- [4] Elfving, R., U. Paulsson, and L. Lundberg, (2002), Performance of SOAP in Web Service Environment Compared to CORBA, Proceedings of the Ninth Asia-Pacific Software Engineering Conference (APSE'02), 2002, IEEE.
- [5] Emerge Project Homepage, <http://www.emerge-project.net/evaluation.htm>, 2004.
- [6] Fattouh, B. and H. H. Saliyah, (2004), Model for a Distributed Telelaboratory Interface Generator, Proceedings of Int. Conf. On Engineering Education and Research, Czech Republic, June 27-30, 2004.
- [7] Hardison, J. D. Zych, J.A. del Alamo, V.J. Harward, et al., The Microelectronics WebLab 6.0 – An Implementation Using Web Services and the iLab Shared Architecture, *iCEER2005*, March, Tainan, Taiwan.
- [8] IEEE Learning Technology Standards Committee, (1999), IEEE 1484 Learning Objects Metadata (IEEE LOM), <http://www.ischool.washington.edu/sasutton/IEEE1484.html>.
- [9] Kenneth Chiu, Madhusudhan Govindaraju, Randall Bramley, "Investigating the Limits of SOAP Performance for Scientific Computing", 11th IEEE international Symposium on High Performance Distributed Computing HPDC-11, 2002.
- [10] Litou, M., (2002), Migrating to Web Services – Latency and Scalability, *Proceedings of Fourth Int. Workshop on Web Site Evolution (WSE'02)*, 2002, IEEE.
- [11] Nielsen, H., J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley, (1997), Network Performance Effects of HTTP/1.1, CSS1, and PNG, <http://www.w3.org/Protocols/HTTP/Performance/Pipeline.html>, June 1997.
- [12] Paquette, G., (1999), Meta-knowledge Representation for Learning Scenarios Engineering, *Proceedings of AIED'99*, Le Mans, France, July, 1999.
- [13] Prolearn Project Homepage, <http://www.prolearn-project.org/>, 2005.
- [14] Robert A. van Engelen, Pushing the SOAP Envelop With Web Services for Scientific Computing, in the proceedings of the International Conference on Web Services (ICWS), 2003, pages 346-354.
- [15] Saliyah-Hassane, H., D. Benslimane, I. De La Teja, B. Fattouh, L. Do, P. Gilbert, M. Saad, L. Villardier, Y. Yan, A General Framework for Web Services and Grid-Based Technologies for Online Laboratories, *iNEER Conference for Engineering Education and Research*, March, 2005, Tainan, Taiwan. (Invited Paper)
- [16] Salzmann, C., and D. Gillet, (2002), Real-time Interaction over the Internet, *Proceedings of IFAC2002*.
- [17] UDDI.org, (2004), UDDI homepage, http://uddi.org/pubs/uddi_v3.htm
- [18] W3C, (2004b), WSDL Specification, <http://www.w3.org/TR/wsdl>
- [19] W3C, (2004a), SOAP Specification, <http://www.w3.org/TR/soap12-part1/>
- [20] Yan, Y., Y. Liang, X. Du, H. Saliyah-Hassane, A. Ghorbani, "Design Instrumental Web Services for Online Experiment Systems", *Ed-Media 2005*, Montreal, June 27-July 2, 2005, Montreal, Canada (accepted).