An Extensible and Interoperable Event System Architecture Using SOAP

Aleksander Slominski, Madhusudhan Govindaraju,
Dennis Gannon, Randall Bramley
Department of Computer Science
Indiana University
Bloomington, IN

Abstract

This paper presents a SOAP-based [6] event system for Grid [8] events, which in turn aid in wiring together distributed software components. SOAP RPC specifies HTTP as its network protocol and XML as the data format; representation of events using XML allows self-describing formats using XML-Schemas, and language and platform independence. We define SOAP events as a specification of interfaces and wire-formats compliant with the SOAP 1.1 specification. We describe the advanced features (directory services, firewall friendliness, security considerations and failsafe mechanisms) that event systems should support. We introduce an interoperable prototype implementation in C++ and Java.

Key Words: Grid, SOAP, Events, XML-Schemas, Firewall, Directory Services

1 Introduction

An event can be defined as a time-stamped message containing typed data that is delivered from a source to a set of subscribed listeners. It may contain additional information for message management, like a sequence number or a time-to-live value.

Event-based systems provide an elegant and robust mechanism to connect software components in distributed computing. They can help in debugging, monitoring, and dynamically negotiating communications protocols in heterogeneous environments. Events can be broadly classified to belong to one of the two categories: *system* events and *application* events. System events typically are used by the component framework to report occurrences such as component instantiation or connection. Application events are for application-specific information like file activity or application state such as "iteration k has completed". An event system should be simple and leverage the benefits provided by existing standards. It should be extensible, language-independent, platform-independent, and provide ready integration with applications that use events internally. There are several scenarios where event services can be useful:

- A process is interested in when a remote process has finished writing a file.

- A drag-and-drop component framework is waiting for successful instantiation of a remote component so that it can be connected to other components.
- A Grid Monitoring System (GMA) [3] needs to collect data for fault detection and performance tuning from a computational Grid.
- Distributed objects need an alternative mechanism to their usual communications mechanisms, for debugging or for (re)negotiating transport protocols for large-scale or specialized data transfer.

This paper addresses the following questions:

- What are the requirements of events in Grid [8] based applications?
- How should event types be defined so that the event framework is extensible?
- What are the *minimal* requirements that publishers and subscribers should satisfy?
- What designs will allow events to work through firewalls?
- Is it possible to design an event system that has built-in recovery mechanisms?
- What is rate at which SOAP based events can be generated and consumed ?

Any new event framework should address five requirements: simplicity, extensibility, rapid deployment and integration, interoperability, and performance. For some time now, it has been clear that systems using XML and SOAP can meet the first three requirements; this paper shows that it can meet the last two as well. In order to specify a minimal standard that satisfies these requirements, we first need to summarize existing ones.

2 Survey of Existing Standards

Any proposed event mechanism must work with existing event standards There is no one best solution. Therefore it is important to have simple and extensible event system that can easily be molded to the needs of applications. Event systems must allow for a naming scheme, object types, security, and leverage Internet standards. Some of the existing standards for event systems can be differentiated according to their design of naming services, push/pull models, capacity to handle firewalls, type description language, language independence, extensions for security, failsafe mechanisms and performance.

2.1 CORBA Events

A CORBA [10] event channel is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events. Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests. References to the event channel may be received from a naming service or from specific-to-task object protocol. CORBA supports a Naming service to locate listeners, and both a *push* and *pull* model. Event types are described using IDL, and the OMG specification describes mechanisms for load balancing and recovery.

2.2 Jini Events

The Jini [14] event system uses the *Jini Lookup Service* for naming which can be optionally used with JNDI [15]. An event is a Java object that can be subtyped for extensibility. The *listener* interface is simple and aids in the use of a flexible *publisher* model. Jini supports *leasing* and uses RMI as the communication substrate. It leverages the built-in security of Java. However, Jini events are designed to work only in the Java environment. Third party objects can handle the distribution of events using *Store and Forward* mechanisms, *Notification Filters* and *Notification Mailboxes*.

2.3 Java Messaging Service

The Java Message Service (JMS) [17] is a Java API that allows applications to create, send, receive and read messages. It defines a common set of interfaces and associated semantics that allows Java programs to communicate with other messaging implementations. JMS provides a loosely coupled architecture that supports asynchronous communication and guarantees reliable delivery of messages.

2.4 ECho Event Delivery System

ECho [9] is an event delivery middleware system. It is designed as an anonymous group communication mechanism. It has high performance event-delivery based middleware that transmits event data in binary. ECho supports the publish/subscribe model of communication and can interoperate with CORBA or Java-based components.

2.5 Grid Monitoring Service Architecture

The Grid Forum Performance Working Group [7] has been studying the performance of systems based on the Grid Monitoring Architecture [3]. They define an event as a structure that contains one or more items of data that relate to one or more resources. In their system an event type uniquely identifies an event and an event schema is used to describe the structure of a particular event. They have defined a set of schemas [4] to represent grid performance using XML-Schemas. The Grid Forum Performance Group plans to use LDAP [12] to interface with a directory service [4].

3 Requirement Specifications

Application and component level events in the Grid environment need to work in heterogeneous environments. Interoperability between disparate systems is a key requirement for distributed component architectures in the Grid. CORBA requires an ORB implementation that conforms to its extensive event specification, while Jini requires the Java environment. It is desirable to draw from these specifications a set of requirements to specify minimal standards for a simple event system. It should be possible to extend this framework to work with CORBA, Jini, JMS or other event systems.

3.1 Requirements for Event-Systems

We can extract from the intended uses and the capabilities of existing event systems the requirements which any event system should satisfy:

- Language and platform independence: The design of an event system should not be based on a particular programming language or a specific environment. It should work with both compiled and interpreted (scripting) languages.
- Interoperability: Some systems in the Grid may be optimized to solve a specific set of problems. An event system should be flexible enough to interoperate with such systems.
- Extensibility: It should be easy to extend the basic event types and add new interfaces to suit the needs of different applications.
- Ease of integration with existing infrastructure: An event system that is based on existing standards like HTTP, XML, JNDI and LDAP can seamlessly integrate into existing applications.
- Lightweight Publishers: Standard libraries providing access to network I/O (sockets) and string manipulation should suffice to publish simple events.
- Simple Listener Interface: The listener interface for even reception should be simple. Such a definition for event sinks provides an ideal foundation for building more sophisticated interconnection of publishers, event channels and listeners.
- Performance: although the speed of any communications framework is dependent upon highly dynamic network environments, creating events and turning them into runtime objects upon reception should take at most a few milliseconds.

3.2 Useful Extensions

Although the previous list specifies minimal requirements for an event system, it should also be easily extended to meet other requirements of complex applications. Most of these extensions are realizable if the event model allows third-party objects to disseminate events. Such third party objects are often implemented as *event channels* which decouple the direct connection between publishers and subscribers. These extensions include:

- Firewall invisibility: An event system should be aware and capable of coping with firewalls.
- Filtering: A listener may be interested in only a specific set of events generated by a publisher. Filtering mechanisms can be put in place to send only those events that a listener has registered interest in.
- Event persistence: A third party object should maintain a list of undelivered events in permanent storage such as disk. This obviates the need for listeners to be connected to the system at the time the event was generated.
- QoS: An event system should provide guarantees about the delivery of events as conforming to at-least-once, at-most-once or exactly-once semantics.

Because a primary use of events is in debugging and long-term performance monitoring, robustness is critical. A leasing mechanism can help maintain this robustness since a leasing-publisher need not keep a persistent list of event recipients. So restarting a publisher need not require restarting all the listeners. Finally, security considerations should be designed into the event framework.

4 Proposed Event System: Grid SOAP Events

SOAP [6] is an object-oriented, Internet based protocol for exchanging information between applications in a distributed environment. The SOAP specification defines the format of data to be in XML. SOAP has the advantage that many programming languages and component frameworks can support it. Since XML is a canonical way of representing data in a tree oriented structure, it lends itself to self-describing and extensible formats. It is to be noted that high performance distributed applications can depend on real-time events and scientific visualization environments often need events with multi-media information. These requirements can be met by highly specialized implementations and cannot be addressed by a generalized framework designed for the Grid environment. SOAP by itself is not efficient for large scale scientific applications but can be effectively used for sending signals and small size information packets in a platform and language independent manner [13]. Thus SOAP seems like an attractive format to specify a event system in a Grid based environment.

4.1 Core Features

The core requirement of Grid SOAP Events consists of a definition for the base SOAP event type and a specification of a SOAP RPC call to an event listener.

4.1.1 Base SOAP Event Type

Figure 1 represents an XML-Schema for a base event type. It draws from the essential features of events in the Jini system. It represents the minimal information that any event is expected to contain. The *eventNamespace* field along with *eventType* uniquely identifies the event. The eventType hierarchy is differentiated with dots: for example an event of type *notebook.experiment.LinearSolver* is a subtype of *notebook.experiment*. The *source* identifies the originating point of the event while the *timestamp* represents the elapsed time in milliseconds since midnight, 1 January 1970. The *seqNo* helps in providing support for different delivery protocols.

The event type is also described using an event schema. Figure 2 shows an example of a resource event encoded in XML. The schema definition for this event derives from the base event type. Figure 3 shows the definition of the XML-schema that provides a formal specification for the derived event type instance. We plan to use RDDL (Resource Directory Description Language) [11] to associate metadata information with SOAP event types.

When an event is received from the wire its type needs to determined so that the values of the elements in the XML payload can be extracted. To achieve this a mapping needs to exist between an XML-Schema and a C++ or Java class in the runtime environment. The

```
<schema targetNamespace = "http://www.extreme.indiana.edu/soap/events" >
  <element name="Event">
    <complexType>
      <sequence>
       <element name="eventNamespace" type="string"/>
       <element name="eventType"      type="string"/>
       <element name="source"         type="string"/>
       <element name="timestamp"      type="long"/>
       <element name="seqNo"          type="long"/>
       <element name="message"        type="string"/>
       <element name="handback"       type="string"/>
      </sequence>
    </complexType>
  </element>
```

Figure 1: The base type event described in XML Schema

```
<MachineUtilizationEvent>
<eventNamespace>http://www.extreme.indiana.edu/soap/events/
                resdat#MachineUtilizationEvent </eventNamespace>
  <eventType>resdata.machine.utilization</eventType>
  <source>rainier.extreme.indiana.edu</source>
  <timestamp>982862312897<timestamp>
  <cpuUtilization>0.88<cpuUtilization >
  <memoryUsed>123988</memoryUsed>
</MachineUtilizationEvent >
```

Figure 2: An example Resource-Event in XML.

event system can then query the class to obtain the values of its fields. If such a mapping doesn't exist for the specified schema type then the event system should throw an exception which in turn should be represented on the wire as a SOAP-Fault element.

4.1.2 Event Listener

Figure 4 shows a simple API for a listener of SOAP events. Note that the type specified in the event is the base type. However, since SOAP supports polymorphism through the XML-Schema Instance type attribute (xsi:type), a derived type can be sent. Mapping events to classes specific to the runtime environment is required only at the receiving end. A run time mapping to the language specific event object can be established as the type of event is read from the xsi:type in the XML payload.

```
<schema targetNamespace="http://www.extreme.indiana.edu/soap/events/resdat/"
                     xmlns:m="http://www.extreme.indiana.edu/soap/events/" >
<import namespace="http://www.extreme.indiana.edu/soap/events/"
  schemaLocation="event.xsd"/>
  <element name="MachineUtilizationEvent" base="m:Event" derivedBy="extension">
    <complexType>
     <sequence>
      <element name="cpuUtilization" type="double"/>
      <element name="memoryUsed" type="long"/>
     </sequence>
    </complexType>
  </element>
</schema>
```

Figure 3: An XML-schema for a specialized event type.

```
interface EventListener {
  void handleEvent(Event ev) throws Exception;
}
```

Figure 4: API for an event listener.

4.1.3 Event Publisher

The generator of events need not have a full implementation of SOAP. An event publisher can just write pre-formatted strings into the socket layer (see figure 5). This makes the proposed event model practical for embedded systems and other applications which need a small footprint, like resource performance sensors [18]. Therefore a formal specification for a publisher is not required, and it suffices to formally state how events should be consumed. Minimal requirements for a simple publisher and listener make it simple to use events in a SOAP-enabled environment.

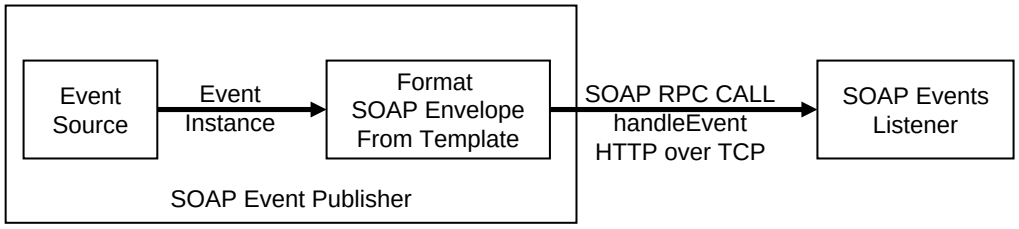

Figure 5: Simple Publisher using pre-formatted SOAP-RPC template for events

4.2 Additional Features

As described in Section 3.2 services should be readily constructed on top of the core SOAP event requirements. A few specialized features such as storing of remote references, leasing and event channels greatly enhance the functionality of a distributed application in which the event system is embedded. For an actual implementation, these specialized features require additional interface specifications.

4.3 Event Channel

```
public interface EventPublisher {
}
public interface EventChannel extends EventListener, EventPublisher {
}
public interface LeasingFilteredEventChannel
       extends EventChannel, LeasingFilteredEventPublisher {
}
```

Figure 6: SOAP API for Event Channel.

An event channel in our proposed system can be used to connect event listeners and event publishers. Figure 6 shows the SOAP API for an event channel.

4.3.1 Remote References

```
<Port>
 <endpoint>
   <location>http://192.168.1.7:4566/urn:soaprmi-v11:leasing-filtered-event-channel
   </location>
   <binding>
    <name>urn:soaprmi-v11:leasing-filtered-event-channel</name>
   </binding>
 </endpoint>
 <portType>
   <uri>urn:soaprmi-v11:temp-java-port-type</uri>
   <name>soaprmi.events.LeasingFilteredEventChannel</name>
 </portType>
</Port>
```

Figure 7: An Example of a SOAP based Remote Reference.

Remote references are handles to objects that reside on remote machines. Direct connections using sockets are not reliable as access may be blocked by firewalls. Remote references

allow communication between publishers and subscribers using proxy servers or direct connections. Remote references can be stored in LDAP and can be used to implement third party objects as is done in Jini.

Figure 7 shows an XML representation of a remote reference. The remote reference contains the location of the remote endpoint (web service) for an event service. The port type that describes the kind of event service provided is uniquely identified by the *uri* of its interface and its *name*. However, the port type may even point to a Web Services Description Language (WSDL) [5] instance. To interoperate with Apache SOAP [1] the URI of the endpoint is also required. The endpoint may have *bindings* that specify how the parameters of the payload should be interpreted. The representation of a remote reference as an XML string makes it easy to store it in a directory service.

4.3.2 Event Publisher: Subscription and Leasing

Although an event publisher can be as light-weight as a simple embedded system inserting text into preformatted strings, robustness may require more utility. When a publisher restarts after a crash, it may not be able to recover the entire list of listeners that were subscribed to it. Persistence of an event channel in the face of failures and location independent mechanism to find new location of event channel is a well known problem. The straightforward solution of each publisher keeping a persistent database of listeners is not viable. For example, "resource sensors" run on host machines and report utilization to remote monitors. Resource sensors are barely-tolerated guest processes on remote machines and must have a small footprint - ruling out using heavy weight persistence.

A widely accepted solution in this case is to use a leasing mechanism. A subscription to an event channel must be periodically renewed so that robustness of the event system can be maintained. To achieve this the event channel grants a lease to its subscribers and each subscriber holds the responsibility of renewing its lease. When the event channel has to be restarted, all listeners eventually resubscribe (when renewing their lease) and event distribution resumes after a temporary lapse.

```
interface LeasingFilteredEventPublisher {
  EventLease subscribeLease(EventListener listener,
                            Event filter,
                            long leaseDuration,
                            String handback)
       throws Exception;
}
```

Figure 8: API for Leasing-Event Publisher.

Figure 8 shows the SOAP API for a Leasing-Event publisher. The subscription call accepts a remote reference as a parameter along with a lease duration. The publisher returns a lease object (see Figure 9) indicating the lease that has been provided. This lease duration may or may not be the same as what was requested. The *filter* argument can be used to indicate filtering requirements. The *handback* argument is sent back to the listener along

```
<schema targetNamespace = "http://www.extreme.indiana.edu/soap/events" >
  <element name="EventLease">
    <complexType>
      <sequence>
       <element name="leaseDuration" type="long"/>
      </sequence>
    </complexType>
  </element>
```

Figure 9: Schema for Event Lease Object.

with the event. The listener can use it to identify the event and use to determine how to react to the event.

```
POST /leasing-filtered-event-channel HTTP/1.0
Host: localhost
Content-Type: text/xml
Content-Length: 1475
SOAPAction: "http://localhost:4566/leasing-filtered-event-channel#subscribeLease"
Connection: Close
```

Figure 10: HTTP header for a subscribeLease call.

Figure 10 shows the HTTP header in a SOAP call to a lease-publisher. The wire representation of a remote reference is shown in figure 11. The SOAPAction header can be required for some SOAP based applications and firewalls, and indicates the intent of the message as to what service it wants to invoke the method on.

4.4 Naming

Event producers and consumers need to find each other on the Grid. The LDAP hierarchical model allows us to provide meaningful names to event channels and model dependencies (see figure 12). Local event channels can connect to higher level group channels which in turn can be connected to an organization level event channel. It is even possible to have a direct connection between event channels at startup time or dynamically at runtime (see Figure 13)

4.5 Firewall

SOAP RPC uses HTTP as the underlying transport protocol. HTTP protocol is "firewall friendly" since port 80 on which HTTP traffic flows is guaranteed to be open in all systems running httpd. The SOAP specification states that data on the wire should be transmitted as plain text. This helps in easy inspection of the data that is received on the firewall-port.

```
<p1 id='id1' xsi:type='ns1:soaprmi.port.Port'
             xmlns:ns1='urn:soaprmi-v11:temp-java-xml-type'>

  <endpoint id='id2' xsi:type='ns2:soaprmi.port.Endpoint'
                     xmlns:ns2='urn:soaprmi-v11:temp-java-xml-type'>
    <binding id='id3' xsi:type='ns3:soaprmi.port.Binding'
                     xmlns:ns3='urn:soaprmi-v11:temp-java-xml-type'>
      <name xsi:type='xsd:string'>urn:soaprmi-v11:simple-listener</name>
    </binding>
    <location xsi:type='xsd:string'>
              http://192.168.1.7:4561/urn:soaprmi-v11:simple-listener
    </location>
  </endpoint>
  <name xsi:type='xsd:string'>urn:soaprmi-v11:simple-listener</name>
  <portType id='id4' xsi:type='ns2:soaprmi.port.PortType'
                     xmlns:ns2='urn:soaprmi-v11:temp-java-xml-type'>
    <uri xsi:type='xsd:string'>urn:soaprmi-v11:temp-java-port-type</uri>
    <name xsi:type='xsd:string'>soaprmi.events.EventListener</name>
  </portType>
  <userName xsi:type='xsd:string'></userName>
</p1>
```

Figure 11: A Remote Reference on the Wire.

4.6 Summary

Grid SOAP event framework specifies simple and minimal requirements for an event and form the foundation for an extensible framework for subscribers and listeners. By specializing the publisher and event channel to provide leasing, SOAP events can be deployed on the Internet and provide robustness in the face of intrinsic failures and crashes of subscribers and listeners. Basing the naming and directory service support for SOAP events on LDAP lends scalability in wide area networks such as the Grid or Internet. To work in both Intranet and Internet environments it is important for an event system to be firewall-aware.

5 Reference Implementation: xEvents

xEvents is a reference implementation of the Grid SOAP Events framework. It is implemented in both C++ and Java. It uses SoapRMI [2] as the underlying communication substrate. SoapRMI layers a C++ and Java based RMI system on top of SOAP RPC, allowing the two languages to interoperate in messaging.

The following are some of the salient features of xEvents:

- Publishers and Subscribers: xEvents contains example implementations for simple publisher and listener. Support for leasing publisher also exists in xEvents.

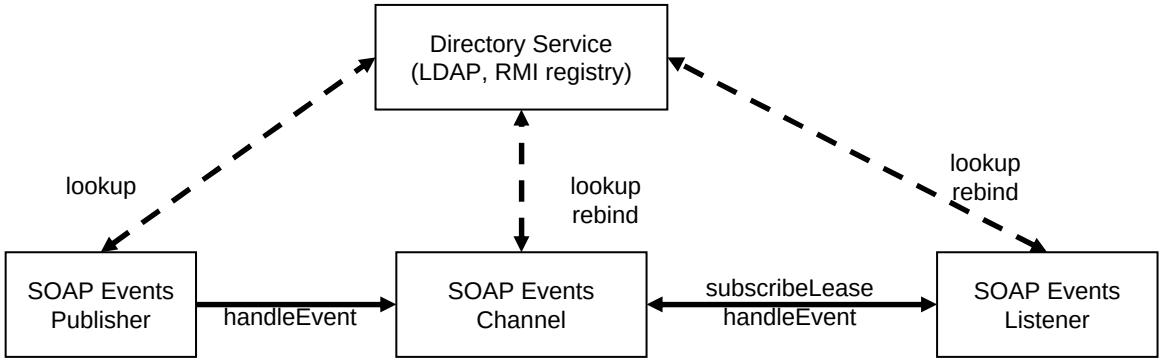

Figure 12: Example of advanced SOAP Events system

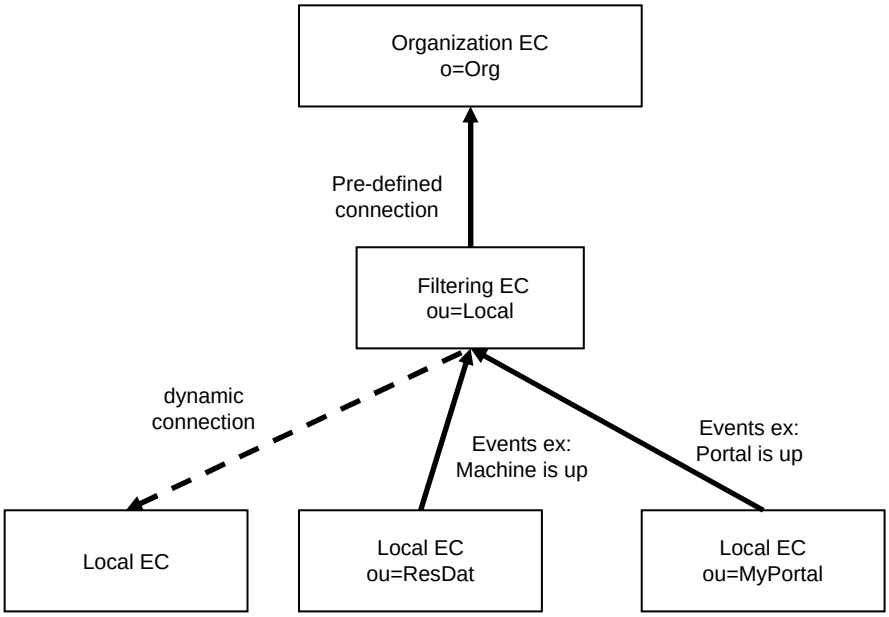

Figure 13: LDAP based hierarchical event channel setup

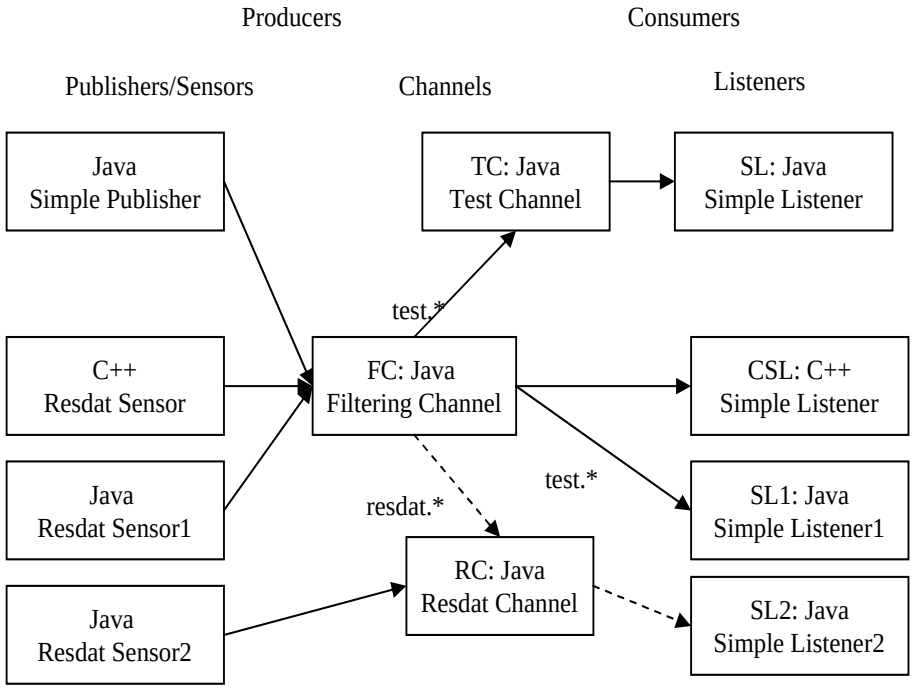

Figure 14: Example of interconnected SOAP Events system

- Event Channel and Remote Reference: SoapRMI has built in support for remote references. xEvents leverages it to transparently pass event listeners as remote references on the wire. An event channel is implemented as a remote interface to support subscription and dispersal of events.
- Naming: xEvents uses the JNDI interface to access LDAP to store remote references. It can also use SoapRMI registries provided by SoapRMI.
- Firewall: To make remote objects such as a publisher or listener residing behind a firewall accessible, xEvents (Java implementation) uses a *tunnel* or *HTTP proxy* that redirects the request to remote objects. Event channels or listeners that reside behind firewalls should bind the URL of the proxy server to the LDAP repository instead of their own IP address. Java provides support for HTTP proxy that is used by xEvents publishers that reside behind firewalls.

6 Examples and Testing with xEvents

We have developed two applications to validate the xEvents implementation.

- Resource Monitoring

  Figure 14 shows a resource monitoring system based on SoapRMI events. ResDat (Resource-Data) sensors in C++ and Java send resource-specific events (CPU utilization, free memory, etc.) to a filtering-event-channel. A simple test publisher sends test-events to the filtering event channel. Other event channels and listeners can be

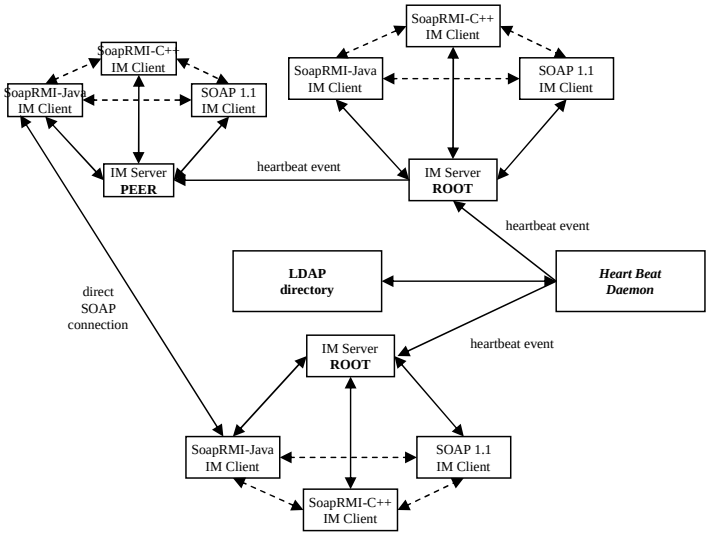

Figure 15: Example of LDAP based Instant Messenger System

connected to the filtering event channel. The filtering-event-channel filters the events (e.g., to select only messages from a specified domain) before it sends it to the test-event-channel and resdat-event-channel.

- Instant Messenger System:

  Figure 16 shows the design of a simple instant messenger (IM) based on SoapRMI events. The clients can be written in either C++ or Java. The server acts as a registry for storing the remote references to the clients. The list of references that the server maintains keeps changing as clients continually exit and rejoin the system. The server even maintains a *buddy* list for each client so that once a client registers itself, all its buddies can be informed. The clients obtain remote references to their friends from the server and can directly chat with them.

  Figure 15 shows the design of a federated instant messenger system. The system consists of interconnected network of IM systems. Clients at boot-up time connect to a local IM server that acts as the root for its subnet. LDAP is used to store the remote references to all IM servers. Whenever a client wants to send a message to another client, it first sends it to its preferred IM server. The IM server searches LDAP to find the preferred IM server for the recipient of the message. If the IM server is online, it sends the message or else the message is buffered.

  This design helps in a simple and efficient mechanism for dispersing events. A central server monitors the LDAP directory and sends a heart-beat event to each root IM server. The root IM server in turn sends the heart beat event to all its clients. If a client doesn't receive a heart beat event in a specified amount of time it initiates the election-process to elect a new IM server as root. All IM servers are expected to rebind to the LDAP directory at regular intervals.

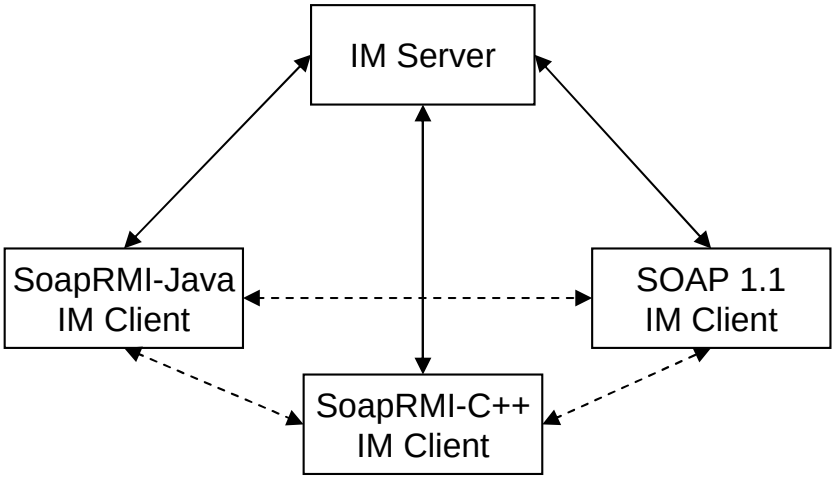

Figure 16: Example of an Instant-Messenger System

7 Performance Measurements

We conducted an initial set of experiments to measure the serialization and deserialization rate for Grid SOAP Events. The tests were done in both the C++ and Java. We measured the performance for a *MachineUtilizationEvent* object, the schema definition for which is shown in figure 2. The size of the serialized form of the object is 1431 bytes. The serialization and deserialization tests did not involve any network communication and so network vagaries were factored out. The tests were done for a number of iterations.

On a Solaris box (UltraSPARC-IIi, 440 MHz, 250 MB memory) the C++ implementation of xEvents serialized at the rate of 9000 events per second. The deserialization rate was 1260 events per second. The Java implementation serialized at the rate of 4800 events per second and deserialized at the rate of 1030 events per second. These results represent the best performance that can be achieved. However in a real application, socket delays, network overhead and implementation decisions (such as use of Java Reflection) can degrade performance.

We plan to further study the performance of xEvents in real applications and answer the following set of questions:

- What is the rate at which Grid SOAP events can be serialized and deserialized?
- What is impact of xsi:type on size and performance of events?
- How does support for namespaces affect the rate at which events can be parsed?
- What is the performance benefit that can be achieved by using per-formatted SOAP strings while communicating with RMI servers ?

8 Conclusions

The minimal design and requirements for simple event systems were developed, one for which more advanced features for complex applications can be readily built. A Grid SOAP events

framework was introduced as a simple and extensible event system. This framework defines an XML-Schema for the base event type and minimal interface for event listener, making it particularly simple.

We proposed the use of event channel with lease based subscriptions and XML based remote references. Leasing helps improve the robustness in the system while developing interconnected network of event publishers (producers), channels (intermediaries) and listeners (consumers).

xEvents is an implementation of Grid SOAP events. The event system in SoapRMI has been implemented in both C++ and Java. This implementation achieves all five criteria specified earlier: simplicity, extensibility, rapid deployment and integration, interoperability, and performance. Our implementation allows direct interoperability between Java and C++, and the performance testing indicates 4500 small events per second can be sent and 1000 can be recieved per second - enough to support the identified Grid applications.

9 Further Work

The current SoapRMI system can be integrated in distributed systems. However we will add security features based on TLS/SSL [16] and digital signatures to provide a secure environment for event communication.

SOAP events are compliant with SOAP 1.1 and as a result are prime candidates to work with applications that use Apache SOAP and .NET framework.

We plan to further study the performance of C++ and Java based event system to determine the bottlenecks in the event system that can be improved. We further intend to add support for archival and persistent event channels.

References

[1] Apache SOAP, visited 02-05-01. http://xml.apache.org.

[2] Aleksander Slominski, Madhusudhan Govindaraju, Dennis Gannon, Randall Bramley. Technical report about design of xml based interoperable soaprmi for c++ and java, visited 03-17-01. http://www.extreme.indiana.edu/soap/rmi/design/.

[3] Brian Tierney et al. White paper: A grid monitoring service architecture (draft), visited 03-10-01. http://www-didc.lbl.gov/GridPerf/papers/GMA.pdf.

[4] Dan Gunter, Warren Smith. Schemas for grid performance events, visited 03-10-01. http://www-didc.lbl.gov/GridPerf/papers/EventSchema.10.2000.pdf.

[5] Erik Christensen et al. Web services description language (wsdl) 1.0, visited 03-12-01. http://msdn.microsoft.com/xml/general/wsdl.asp.

[6] D. Box et al. Simple Object Access Protocol 1.1. Technical report, W3C, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[7] Grid Forum. Grid performance working group, visited 03-01-01. http://www-didc.lbl.gov/GridPerf/.

[8] Ian Foster and Carl Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[9] Georgia Tech. The echo event delivery system, visited 03-12-01. http://www.cc.gatech.edu/systems/projects/ECho/.

[10] Object Management Group. The Common Object Request Broker: Architecture and specification, July 1995. Revision 2.0.

[11] Jonathan Borden et al. Resource directory description language (rddl), visited 03-12-01. http://www.rddl.org/.

[12] M. Wahl et al. The lightweight directory access protocol (v3), visited 03-01-01. ftp://ftp.isi.edu/in-notes/rfc2251.txt.

[13] Madhsudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, Dennis Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of SuperComputing 2000, Dallas TX, 2000*, November 2000.

[14] Sun Microsystems. Jini, visited 3-1-2001. http://www.sun.com/jini.

[15] SUN Microsystems. JNDI, visited 3-7-2000. http://java.sun.com/products/jndi/.

[16] Network Working Group. The transport layer security protocol, visited 03-01-01. ftp://ftp.isi.edu/in-notes/rfc2246.txt.

[17] SUN Microsystems, Inc. Java message service api, visited 03-12-01. http://java.sun.com/products/jms/.

[18] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 1999. also UCSD Technical Report Number TR-CS98-599, September, 1998.