

# XEVENTS/XMESSAGES: Application Events and Messaging Framework for Grid

Aleksander Slominski  
Yogesh Simmhan  
Albert Louis Rossi  
Matthew Farrellee  
Dennis Gannon

Indiana University Computer Science Department \*

## 1 Introduction

Computational Grids have gained dominance in large scale scientific and engineering research. They enable researchers to collaborate and solve problems by providing seamless access to pervasive collection of computing resources.

One of the requirements in Grid computing is the need to monitor interconnected components that are part of a distributed scientific application. Event based systems pose an elegant and robust solution to make this possible.

An event can be described as a message containing typed data generated by a source and delivered to any listener interested in events of that type. It may contain additional information like timestamp or sequence number used by the application or used for message management.

Events can be broadly classified into two categories: lower level events such as middleware or system events and higher level events like application events. The system events are typically built into the services layer of the framework being used by the Grid application. For instance, a component may generate an event upon its instantiation or connection to another component. Events used to monitor network or host performance, as in the case of Network Weather Service (NWS), also constitute middleware events. Application events disseminate application-specific information like file activity or application state transition.

In this paper, we address the following research questions:

1. How does one design a Grid-based event system that is compatible with web-services?
2. What are the design features we require to make the event delivery robust and reliable?
3. How can existing standards like SOAP/XML be leveraged to our advantage?
4. Is the system extensible and flexible enough to interoperate with diverse systems?
5. Can we construct an API simple enough to use without constricting the functionality?
6. What additional features, such as filtering and logging, would be useful to the user?

In addition to satisfying the answers posed by the above questions, it would increase the credibility if such an event system has been deployed and used with distributed applications. We shall demonstrate the design of such an event system and exhibit its use in Grid applications.

## 2 Survey of existing systems

There are a number of event systems available. We briefly describe some of them and highlight the features that are necessary and those that are lacking in the existing systems.

---

\*Department of Computer Science, 150 S Woodlawn Avenue, Bloomington, IN 47405. Ph: 812 855 8305 Fax: 812 855 4829

## 2.1 CORBA Events

A CORBA event channel is an intervening object that allows multiple suppliers and consumers to communicate 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 has both a *pull* and *push* model. Event types are described using an Interface Definition Language (IDL) and the OMG specification describes mechanisms for load balancing and recovery. The quality of service depends on the implementation of the event channel and the reliability provided may vary from “at most once” delivery to a “best effort” policy. The implementation also determines the order in which the events are received.

## 2.2 Jini Distributed Events

Distributed events in the Jini framework allow an object in one virtual machine (VM) to register interest in the occurrence of an event in another object, possibly running in another VM, and receive notification when such an event happens. The Jini Event System uses the *Jini Lookup Service* for naming which can be optionally used with Java Native Directory Interface (JNDI). 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 Java-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 and are not equipped to work with firewalls and Network Address Translation (NAT). Third party objects can handle the distribution of events using *Store and Forward* mechanism, *Notification Filters* and *Notification Mailboxes*. The protocol does not indicate the reliability and timeliness of the notifications and is left to the implementation of the various objects.

In designing a distributed event system, it is desirable to have a programming language independent solution since the Grid contains heterogeneous applications written in various languages. This means that the representation of the data on the wire also needs to maximize portability. It is widely accepted that XML is an ideal choice for platform and language independent representation of data and as an extension, SOAP is preferred for invoking remote procedure calls (RPCs). The Web services community has also gone in with this.

## 2.3 Java Messaging Service

The Java Message Service (JMS) is a Java API that allows applications to create, send, receive and understand 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. The specification provides for both point-to-point messaging using queues and the publisher/subscriber approach using topics as intermediaries. Messages can be consumed both synchronously (“pull”) and asynchronously (“push”). It also has message filtering capabilities in the form of message selectors based on a subset of SQL92 conditional expression syntax.

However, JMS does not specify how events are represented on the wire. As a result, interoperability between different JMS providers is not ensured and the primary intention is to provide a standard messaging API.

## 2.4 ECho Event Delivery System

ECho is an event delivery middleware system. It is designed as an anonymous group communication mechanism. It has an efficient event propagation middleware, the event channel, which achieves good performance characteristics due to the use of a binary protocol for event transmission. ECho supports the publish/subscribe model of communication and can interoperate with CORBA and Java-based components.

In spite of these features, interoperability is lacking due to the use of a custom binary protocol which other messaging systems will have to use in order to be compatible with it.

## 2.5 Grid Monitoring Service Architecture

The Grid Monitoring Architecture is a working group of the Global Grid Forum and it is focused on producing a high-level architecture of the components and interfaces needed to promote interoperability between heterogeneous monitoring systems on the Grid. Events are defined to be timestamped sets of information with a type. Means of interaction between producers and consumers, that are being considered, are “publish/subscribe”, “request/response” and “notification”. The wire protocol used to transfer events is not specified and neither are delivery guarantees mentioned. A Grid Information Service (GIS) is used to maintain an event dictionary. In addition, the producers register themselves with the GIS and the consumers can search for producers registered with the GIS. The Grid Forum is in the process of adopting LDAP as their standard for information services.

Nonetheless, the GMA is interested in events for performance analysis and problem monitoring and the requirements for it vary from that of application level events.

## 2.6 SoapRMI Events

SoapRMI Events is our previous version of an application level messaging system for distributed environments. SOAP is used as the format for specifying the events and invoking the RPC calls. An event is uniquely identified by its *namespace* and *type*. A *source* field is used to indicate the originating point of the event. A *timestamp* is also present along with a *message* field for including additional information. A *handback* field can be provided by the listener to the publisher and this is set when returning events to the listener.

An event publisher and event listener, that produced and consumed events respectively, are defined. An event publisher can generate events and publish it by just writing preformatted SOAP string into the socket layer. This gives it a light footprint and makes it useful for embedded systems and resource monitors, other than regular grid applications.

One of the problems with the publisher is the handling of exceptions in case an event cannot be sent due to a network glitch or the listener being temporarily unreachable. An exception is thrown by the publisher and the application has to handle it. This puts an additional burden on the application program writer to handle failure cases. In addition, the event send is a blocking call to enable synchronous failure notification and this overhead becomes significant when many events are sent. We have overcome these limitations by using a non-blocking send in a separate thread of execution, coupled with a logging mechanism that enables retrying sending the events in case of an error. We use an intermediary known as publisher agent for this purpose.

Retrieving events in SoapRMI events is based on leasing. An event listener subscribes to a publisher with its remote reference, the lease duration and the type of events it is interested in. The publisher uses the remote reference of the listeners to send events to all those registered with it for a particular event type it has for publishing. This leasing approach is used to reclaim resources in case a listener fails to unsubscribe. An automatic lease renewal system, that enables the lease to continue as long as the application exists, is also available.

This approach is elegant and works well on a small LAN but does not scale effectively on a large scale Internet environment. The problems are due to two factors: (1) not all clients have static or globally accessible IP addresses and (2) network and server outages are fairly common in a large network.

Every listener's remote reference is a URL that includes the IP address of the host on which it is running. These are typically dynamic IP addresses and, in combination with firewalls, are inaccessible outside the local network. So a listener with such a remote reference cannot have events pushed to it when subscribing to a publisher outside the local network. This is overcome by allowing an alternative way of retrieving events by “pulling” them from the publisher (i.e.) the listener initiates the retrieval by contacting the publisher instead of the publisher having to use the remote reference of the listener.

The latter problem relates to robustness in the event of a network failure. If a listener's lease expires during a network outage, the publisher discontinues retaining the events for that listener since the lease is not renewed. When the resubscribe finally gets through, the listener has lost events that took place in the interim between the connection being lost and the resubscription occurring. One way to manage this is for the publisher to save the events in a persistent store so that the events are retained. In this case, the listener can use the timestamp of the event to select the ones to retrieve. But it is still possible to retrieve duplicate

events or miss some of them due to the nature of the timestamp. This problem is surmounted by having the mechanism to have unlimited or persistent subscription. The listener is granted the persistent lease by an agent while the agent takes care of maintaining a valid lease, of limited duration, with the channel. Also, each event retrieval is assigned a permanent id that encapsulates the state information so that using that ID always returns the same set of events from the channel along with the ID to be used for the successive set of events. So, even if a lease is lost, events can be retrieved from where they were left off.

### 3 Events and Messaging: Desiderata

From the above discussion, we can arrive at a desiderata of features for an event system that intends to cater to distributed applications in the grid:

1. Language and platform independence: The design of an event system should not be tied to a particular programming language or a specific environment.
2. Interoperability: An event system should be flexible enough to interoperate with other messaging systems present in the grid.
3. Extensibility: It should be easy to extend the basic event and add new interfaces to suit the needs of different applications.
4. Ease of integration with existing infrastructure: An event system that is based on existing standards like HTTP, XML and LDAP can seamlessly integrate into existing applications.
5. Simple and flexible interface: The application writer must be provided with a simple API to publish and retrieve events as it would provide a foundation for building more sophisticated interconnections of the objects. It should also be possible to initiate event retrieval by the listener (“pull”) or by the publisher (“push”).
6. Performance: Although application events are not expected to be large and frequent, serialization and deserialization should be efficient.
7. Reliability: An event system should provide atleast a “best-effort” guarantee about the delivery and reception of the event since most application events are non-trivial. It would also be useful to be able to tune this to “atleast-once”, “utmost-once” or “exactly-once” semantic depending on the application.
8. Web Services: Web Services is gaining wide spread importance and it would be advantageous if the event system exposed a Web Service interface to enable external and possibly unknown applications to make use of the service.

Other useful extensions to meet more complex application requirements include:

1. Firewall invisibility: An event system must be capable of coping with firewalls.
2. Filtering: In many cases, the listener may be interested in only some of the many events being published and it should be possible to filter the events based on various fields present in the events.
3. Persistence: It would be useful to be able to store the events on a disk or database to make it possible to perform historical queries during future analysis of the events.

### 4 Architecture and Implementation

We have followed a layered approach while developing the XEvents/XMessages framework [?]. The application program lies at the top of the layer and uses the event listener and publisher interfaces. The event listener and publisher can interact with each other directly, through an event channel (as in the case of SoapRMI-Events) or use the XEvents agent middleware comprising of the event listener and publisher agents. They ensure exception handling and provide a wrapper for the lower level messaging layer. The XMessages agent middleware enhances reliability by maintaining a log of messages that are to be sent so that none of them are lost if they cannot be sent immediately. It also provides automated lease renewal with the channel on behalf of higher level layers. The message channel has sink and source interfaces for accepting and forwarding messages, in addition to storing the messages in a persistent media for querying. Messages may be defined as any XML content that needs to be transmitted between source and sink while events are also XML strings but have the typed fields we discussed previously in SoapRMI-Events. Overall,

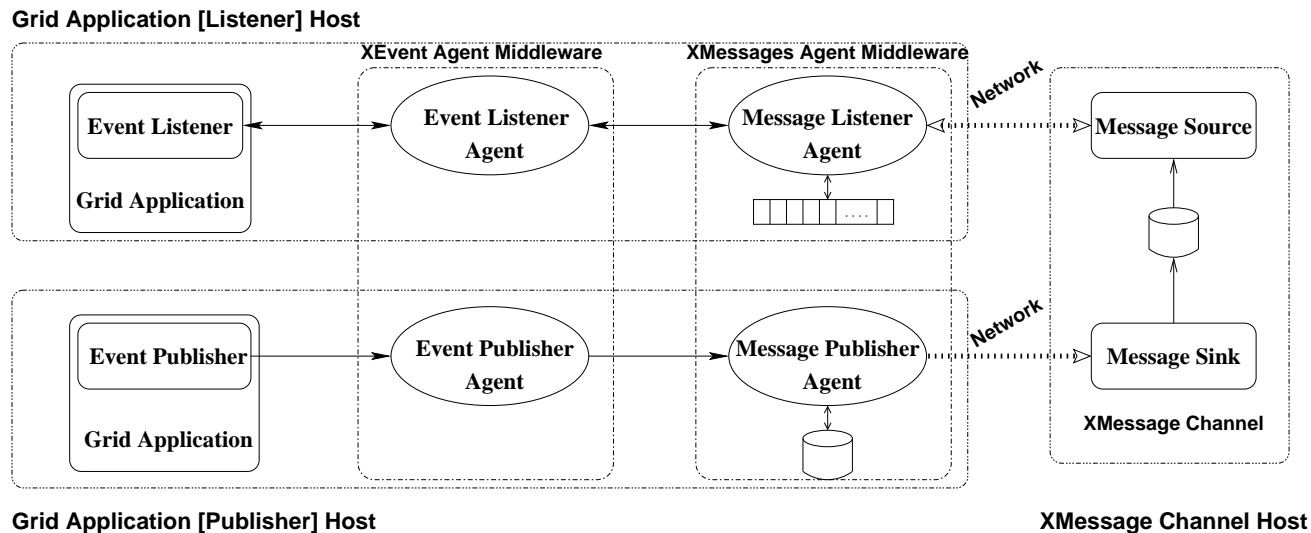


Figure 1: Architecture of XMessages/XEvents framework

we can discern the XMessages API providing dependable messaging service and the XEvents system that uses this for transmitting events.

## 4.1 XEvents

XEvents provides a simple API for publishing and listening for events. It defines interfaces for event publishers and event listeners which grid applications use for exchanging events. A publisher is the source for an event and a listener is interested in consuming these events. A publisher and a listener can either interact directly with each other, exchanging events or use an XMessage channel that allows multiple publishers and listeners to communicate asynchronously. The interaction with the channel is achieved through agents which provide value added services.

As in SoapRMI-Events, we continue to use SOAP calls for method invocation during messaging and XML is used to describe the events. We have an implementation of SOAP, know as XSoap, which is used. We also retain the structure of the events fields as is used in SoapRMI events since they continue to serve the purpose adequately.

### 4.1.1 Publishing Events

Publishing an event can be as simple as getting the reference to a listener and sending preformatted XML to it by opening a socket connection to it. The listener defines a *handleEvent* method which is invoked using a SOAP call. But, as we saw in SoapRMI-Events, such an invocation has the disadvantage of making the application programmer take care of exceptions that may occur if the listener cannot be contacted or if it is not available while sending the event.

```
EventListener listener =
    xevents.xsoap_impl.lookupEventListener(url, context);
...
try{
    listener.handleEvent(myEvent);
} catch(Exception ex){
    ...
}
```

So we introduce a publisher agent that acts on the behest of the event publisher. The publisher agent implements the event listener interface, the difference being that it does not throw an exception and neither does it block on the *handleEvent* call. The publisher creates an XEvents publisher agent and associates it with the channel to which it is sending events. The XEvents publisher agent is just a wrapper for the XMessages publisher agent and does not provide any value added features. It is present to retain the symmetry of the layers.

```
// Create a publisher agent from the
// (default) XSoap implementation
EventPublisherAgent pAgent =
    XEventsImpl.getDefault().createPublisherAgentFor(location);
...
// Send the event to the publisher agent
pAgent.handleEvent(myEvent);
```

The XMessages publisher agent has two main functions. One is to log the event and the other to enable non-blocking sending of the event. The logging interface allows for storing the XML content of the *handleEvent* SOAP call invoked on the channel by the agent. It also keeps track of the status of the event as pending, sent successfully or error in the send (caused by the event being rejected by the channel). The default log is maintained in the file system by using a message file that has the SOAP RPC's content and an index file with the location of each event in the message file, its size and status. The events that have been sent successfully are removed while garbage collecting. It is possible to inspect the store to check the status of each message at a later stage. Events from the log can be delivered to their destination even after the application has finished executing. Such postponed sending of events can be accomplished with a simple utility that reads the event log and sends unsent events.

The second functionality is one of 'fire-and-forget' non-blocking call of the *handleEvent* method by the application. This is achieved by using a separate thread for sending the events as they are received from the application and have been stored in the log. This uses a mechanism similar to the sliding window protocol in TCP. The events to be sent are stored in a queue with two pointers denoting the last event sent (LES) and the last acknowledged event (LAE). The LES is incremented as events are sent and the LAE incremented as the acknowledgments for the receipt of the event is received from the message sink. In case an exception occurs, the LES is reset to LAE+1 so that all events since the last event acknowledged is resent. This causes duplicate events to be transmitted. So the message sink has to have the ability to detect and remove duplicates.

Since the publisher agent uses additional threads for sending of events, the *handleEvent* call is non-blocking. Therefore, the events are sent in the background and the overhead associated with the application waiting for the events to be sent is minimized. Moreover, there is no exception to check for - the publisher agent handles event sending and will log errors while keeping unsent events in a persistent event log.

Other than the Java implementation, we provide an implementation of the event publisher and agent libraries in C++ too. The C++ agent can be created as shown below. This enables applications written in C or C++ to publish events to an XMessage channel.

```
/* Return the XSoap (default) implementation */
XEVENTS_XEventsImpl* eventImpl =
    XEVENTS_XEventsImpl_getDefault();
/* "NOWAIT" is default */
eventImpl->setTimeoutAtExit(XEVENTS_WAITFORTHREADS);
/* Get the non-blocking, persistent agent */
pAgent
    = eventImpl->createPublisherAgentFor(eventImpl, locationUrl);
...
/* Send the event to the publisher agent */
pAgent->handleEvent(pAgent, myEvent);
```

### 4.1.2 Retrieving Events

Applications can retrieve events by implementing the event listener interface. The retrieval can be through “pull” or “push” or a combination of the two. In a “pull”, the events are fetched from the publisher upon initiation by the listener. In a “push”, the listener subscribes to the publisher for events matching some filter and the publisher sends events to the listener as and when events of the requested type are generated. The push model is well suited for delivering “live events”, as and when they happen. But they have the drawback of the publisher having to keep the state information and listener location with it. So, if the publisher goes down and is brought up again, this information is lost and any events that took place in between are also missed. Also, we have the problem of the listener possibly being behind firewalls. In such a case, the pull model is the preferred alternative. But the pull model necessitates the application to periodically contact the publisher, possibly in a loop, for retrieving successive batches of events. In either case, the publisher will have to have a persistent store of the events in order to enable the listener to receive all events in the face of protracted network failures.

As we saw in SoapRMI-Events, the key problems are caused due to network failures while subscription is to be renewed and the requirement of the publisher having to maintain state information about the listener. These are addressed by using the listener agent middleware. The listener agent duplicates the interface exposed by the event publisher so that the listener can invoke methods just as it would on a publisher. The XEvents listener agent is essentially an adapter between events and messages and acts in tandem with the XMessages listener agent that provides the prime functionality.

```
// declare listener for events
public class MyListener
    implements EventListener
{
    public handleEvent(Event ev) throws Exception {
        System.out.println("received even from "+ev.getSource());
    }
}
EventListener eListener = new MyListener();
...
// get handle to event publisher
EventListenerAgent lAgent =
    XEventsImpl.getDefault().createListenerAgentFor(location);
EventSubscriptionLease lease = lAgent.subscribeLease(eListener);
try {
    // Sleep for 10 seconds while events are pushed
    Thread.currentThread().sleep(10 * 1000);

    // Switch from push to pull for next 10 seconds
    agent.renewSubscription(lease, 10 * 1000 /* duration */ );
    Event[] events =
        lAgent.pullEvents(lease,
            10, /* how many events to pull */
            10 * 1000 /* timeout */ );
    ...
} finally {
    lAgent.cancelLease(lease);
}
```

The application initially creates an XEvents agent that internally starts an XMessage to XEvents adapter for the events retrieved and an XMessage listener agent. The client application starts a lease with the publisher. The agent in turn negotiates a remote lease with the publisher for the requested or maximum allowed duration and returns a local lease for the requested duration to the client. The agent then makes sure that the lease with the publisher is kept valid by resubscription as long as the applications maintains

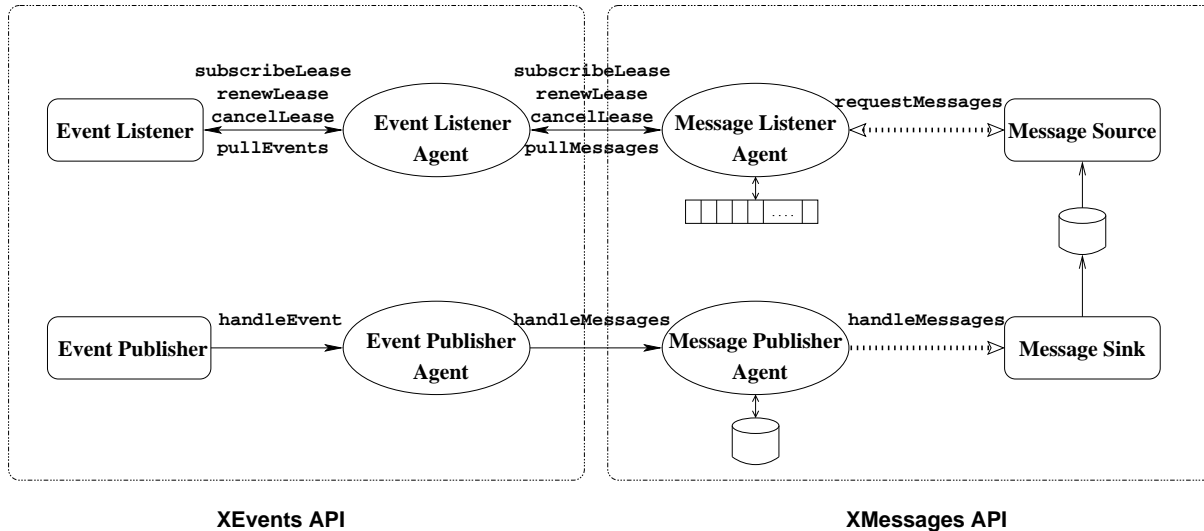


Figure 2: API of XMessages/XEvents framework

the local lease. It also begins to cache events by starting a thread that pulls events at regular intervals from the publisher. If the client has requested for a push subscription, then the agent sends events that it pulls from the publisher to the listener. If the client has a pull subscription and invokes the pull method, the agent returns the matching events from the cache. As an additional feature, it is also possible to dynamically switch between a push model and a pull model seamlessly without losing any events in between.

In the example given above, a push lease is initially subscribed to for the first 10 seconds. Events that take place during this time are sent to the listener by invoking the *handleEvent* callback on the listener with the events. Then the subscription is changed to a pull model by renewing the lease to reflect this. The *pullEvents* method is called with the maximum number of events to return and the timeout for which to wait for events. This allows the listener to throttle the number of events it pulls from the publisher at one given time. Successive calls to the method will return the next set of events. Once the application is finished with retrieving the events, it cancels the subscription so that the agent may free up resources it has allocated for the listener and release the lease it has with the publisher.

Thus, in both pull and push subscription mechanisms, the agent uses the pull model for retrieving events from the publisher. This overcomes the problem of dealing with firewalls. The caching also improves performance for pulls but when push subscription is used, the events are sent to the listener in pseudo-realtime i.e. not as they are generated but when the agent does a periodic pull on the publisher. In addition, the listener is also able to get a persistent subscription with the agent for as long as it requires since the agent manages the subscription to the publisher to match the needs. If the publisher cannot be contacted, the agent retries connecting to it and retrieves events when the subscription is reestablished. Hence, the entire process of establishing and maintaining a connection with the publisher in the face of network failure is kept transparent to the listener and managed by the agent.

## 4.2 Generic XMessages API

The XMessages API provides a generic messaging platform with WSDL support, on top of which XEvents is built. Other than the XMessages listener and publisher agents which have been discussed above, the API defines the XMessageSource and XMessageSink interfaces and two structures, XMessageRequest and XMessageBatch, for requesting and returning messages. The WSDL port types for the interfaces and XML schema for the structures are listed in the XMessages/XEvents whitepaper. These can be used for accessing the messaging framework as a web-service. The WSDL files describing port types and XML schemas for message containers are available at <http://www.extreme.indiana.edu/xgws/xevents/v10/xml/>

#### 4.2.1 Structure: XMessageBatch

XMessageBatch is the basic unit of data exchange in XMessages. We wrap a set of messages into this auxiliary structure, in addition to fields that describe the messages. We could leverage SOAP to include this metadata in the SOAP header but having one object encapsulate both data(messages) and metadata makes it easier to transport these entities through multiple protocols.

XMessageBatch contains the following fields:

- **Object[] messages**  
Set of messages represented as an array. We do not do make any assumptions about the kind of object that the message is, as long as it can be serialized to the chosen wire protocol.
- **String currentToken**  
If the value is not null, this acts as a globally unique token that identifies the current set of messages and can be used to avoid processing duplicate messages.
- **String nextToken**  
If the value is not null, this token indicates that a subsequent batch of messages is available and identifies it. *nextToken* can be passed to the XMessageSource to retrieve the next message batch. The token defines a state in the message source; so by returning this to the message sink, it makes it possible for the source to avoid maintaining state information.
- **int leaseDurationInMinutes**  
The duration (in minutes) for which the session information pertaining to this batch retrieval will be cached. Requesting further messages using the same lease beyond this time will incur additional resources and processing. Message sources that retain messages persistently should be flexible with regard to the enforcement of lease duration by allowing message retrieval after the lease expiration, sometimes even after days. Non-persistent message sources may return an error indicating that the session lease has expired and should be renewed. The actual behavior in both cases may vary according to the implementation.
- **long timeToLiveForMessagesInMs**  
The period (in milliseconds) for which the messages are to be used. -1 implies the message should be kept for as long as possible while a 0 means the message should be discarded after processing. This can be used to determine the duration for which the messages are cached.

#### 4.2.2 Structure: XMessageRequest

XMessageRequest is used to encapsulate the parameters required by the message source for message retrieval. XMessageBatch contains the following fields:

- **Object query**  
The query object determines which messages will be selected for retrieval. To allow extensibility, this query is a generic object whose interpretation depends on the message source. For example, it could be an SQL query string that is understood by a particular implementation of the message source. The query is overloaded as a token and it returns the messages associated with the token. Using the same token returns the same set of messages.  
  
While monitoring live messages i.e. messages that are being retrieved as they are generated, if the request does not return any messages, then the same token needs to be sent. Otherwise, the *nextToken* from the XMessageBatch should be used to get the subsequent messages batch.
- **int timeToWaitForMatchInMs**  
This controls query execution in case no match is found immediately. The message source waits for *timeToWaitForMatchInMs* milliseconds for the query to get messages before returning.
- **int maxNumberOfMessagesToReturn**  
When the query matches a large number of messages, it would be prudent to get them in smaller sets than sending them in a single batch. This field controls the maximum number of messages returned in the message batch. If 0, it indicates using the default specified in the message source.

- **long maxTimeRangeInMsToReturn**

One of the parameters used while pulling messages is the time at which the message was generated, if available. This argument determines the maximum range of time (in milliseconds) allowed between the first and last matching messages returned. If 0, this field is ignored.

- **long suggestedLeaseDurationInMinutes**

The duration (in minutes) for which the client wants a lease to be granted. The actual lease granted is returned in the *leaseDurationInMinutes* field of *XMessageBatch*.

- **String subscriberWSDL**

Passing the WSDL of the subscriber requesting pushing of messages allows the message source to send the matching messages directly to the message sink. If null, the message request will be taken as being in pull mode.

- **int maxNumberOfMessagesToPush**

While *maxNumberOfMessagesToReturn* controls the maximum numbers of events to be returned in pull mode, the equivalent in push mode, *maxNumberOfMessagesToPush*, determines the maximum number of messages that may be pushed in one batch by message source to message sink

- **boolean takeMessages**

If this is true, the messages will be removed from the message source when they are returned to the message sink and will not be available for other message requests.

#### 4.2.3 Interface: XMessageSink

This interface defines how messages can be consumed. It contains a single method, *handleMessages*, that must be called in order for the *messageBatch* to be consumed. The way in which *XMessageSink* processes the messages in the *messageBatch* depends on the implementation; it could handle the messages and forget them, it could forward it to one or more *XMessage* sinks or store them in a database for future retrieval. If an error occurs while handling the message batch, an exception is returned.

```
public void handleMessages (XMessageBatch messages) throws Exception;
```

#### 4.2.4 Interface: XMessageSource

The *XMessageSource* interface either produces messages itself or has messages available through other means for retrieval. Messages can be retrieved through the *requestMessages* method that initiates, continues and terminates the process of requesting the messages. The *XMessageRequest* parameter is used to determine the messages that will be selected for returning in the message batch. The *query* field in the message request argument is interpreted either as a token, in which case the messages corresponding to the token are chosen, or as a generic query, which is processed as defined in the implementation of the message source. In both cases, the message batch is populated with the messages matching the query, tokens identifying current and next set of messages, and other fields present in the message batch object. Further messages can be requested using the *nextToken* field for the message batch as the query string in the message request parameter. The session for this request operation can be cancelled by passing a lease request with zero duration.

```
public XMessageBatch requestMessages ( XMessageRequest req ) throws Exception;
```

#### 4.2.5 Message Channel

The *XMessageSource* and *XMessageSink* can be grouped together to form an intermediary object known as the message channel. This channel can provide a host of value added services like multiplexing and demultiplexing messages between multiple listeners and publishers, maintain messages in a storage media to enable querying and analysis of past messages, provide ubiquitous messaging service to a cluster of related grid applications or connect a number of message channels to form a large event system. Such an intervening object makes it possible to extend the features of the messaging system to meet the characteristic needs of diverse grid applications.

In the current implementation, we have used the channel to save messages in a database to enable simple querying of past messages and to detect duplicate messages that may be sent by the publisher agents. The

current and next tokens are used to ensure the same message is not processed again even if it is sent again. Other useful additions that could be made are providing hooks for triggers that match a certain criteria of messages, do accounting and gathering statistics about messages that pass through and act as bridges to other messaging systems.

## 5 Salient Features

In this section, we will explore the set of features present in the XEvents/XMessages framework that make it an effective grid messaging system.

### 5.1 Reliability and Robustness

The need for a reliable event service that handles most of the failure cases expected in a complex grid environment is understood and addressed by the XEvents/XMessages system. Since it is likely that events would be used to a certain extent to inform other components of errors while executing a particular application, it is important that the event service does not introduce exceptions of its own that require the intervention of the application program. In order to ensure this, the various possible failure cases in the messaging framework have to be ascertained and their occurrence prevented.

In order to exchange messages, publishers and listeners need to be able to contact each other. If the network is down and they are unable to be reached, the network connection must be tried repeatedly till it is up and running. By using the agent middleware, this process is automated. If the network goes down while messages are being transmitted, messages should not be lost. To assure this, the publisher agent logs the message before sending it so that the message can be recovered in case the network fails while it is being sent. Similarly, the listener agent uses persistent leases and the tokens to make sure all messages are retrieved.

If a message sink is unable to process a message and throws an exception to message source, the publisher agent traps the exception and logs it. The publisher agent also sends the events in the same order that it has been received from the application, though it may send duplicate messages at times when the message may have been sent but if the acknowledgment is lost. The tokens in the message batch are used to remove these duplicates.

These features ensure that no messages are lost and that a “best effort” will be put to assure delivery while the network is up.

### 5.2 Filtering and Querying

Numerous events may get generated by a publisher and usually, the listener is interested in only a subset of these. So it is necessary to have filtering capabilities that select only the events matching the requirements of the listener, for retrieval. Simple filters may match certain fields of the message while others may perform more complex tasks like executing queries similar to SQL's. To enable such varied use of the filter that would depend on the particular needs of the application, a generic query object is passed in the *XMessageRequest* structure. This query can be understood as required by customizing the implementation of the message source.

One of the features critical for a successful event system intended for the scientific applications is the ability to store application generated events permanently and allow users to execute queries on them. Such historical queries can be used to discover application behavior under different circumstances. The query object mentioned above can enable the user to perform such data mining by allowing retrieval of both historical and live events using a consistent message request interface. In addition, by using an RDBMS to store the events in the back end, we have provided the user with the option to use sophisticated data mining tools. We have provided initial support for SQL like queries as the persistent message storage is currently based on an RDBMS. Other querying mechanisms, in particular simple template queries similar to JINI, will be added in future.

Another factor that helps in performing historical queries that tend to be large is the ability to retrieve chunks of the message set and not the entire list of matching messages. This incremental retrieval allows the application to cancel the query midway without having to process all messages originally asked for.

### 5.3 Flexible, Extensible, Interoperable

The messages that are transmitted over the XMessages framework are represented as XML on the wire. This gives the flexibility to interpret them generically and extend them to different messaging domains. Since the method invocations use SOAP, it is easy to write bridges that convert between other messaging APIs to the XMessages framework so that a message sink from another messaging implementation can retrieve messages from an XMessages source or vice versa. This makes it well suited to operate in a heterogeneous grid environment which may require interaction with other messaging systems.

Another point in favor is the possibility of pulling messages, to simulate a push model, using the listener agents. This allows listeners behind firewalls to contact message sources and retrieve messages from them. Similarly, publishers behind firewalls can send messages to message sinks that are visible. Using this feature, we can create an entire messaging system by having just one host running the message channel (comprising of a message source and sink) to be visible and all other listeners and publishers can be protected by firewalls.

When building messaging environment that spans across multiple clusters, it helps to have many message channels that can service an optimal set of hosts and yet have the ability to interchange messages with listeners and publishers present beyond the local channel. This can be achieved by interconnecting message channels to form a messaging network or a cloud. In such a system, each channel would act as a listener to the other channels and subscribe to remote messages. We would need to have a proper message passing algorithm to ensure we do not have runaway messages that propagate indefinitely and avoid storing duplicate messages in the persistent store. Having a globally unique token for each message batch helps in addressing these concerns.

### 5.4 Simple API

The XMessages framework has a light weight, yet powerful API that makes it possible to perform tasks ranging from rudimentary messaging using just a preformatted string writer as message source and a simple message sink, to a full featured messaging service that allows for querying and interconnecting multiple listeners and publishers. As we have shown with the XEvents extension, it is possible to build additional capabilities to the XMessages API and implement other messaging systems like JMS using it.

### 5.5 Grid Web Services

We have provided support for accessing the XMessages grid messaging framework as a web service by specifying the WSDL port types for the methods and the XML schemas for the data structures. This allows other applications that are not in the local environment to use the messaging service over the internet. It also does not tie one down to a single programming model or language for using the system.

## 6 Further Work

There is potential for extending the current design to provide more functionality. We describe some of the further work that is in progress to expand the features of the framework.

### 6.1 Enhanced querying support

The SQL querying available currently is sufficient for basic retrieval needs. But there is scope for adding other types of query objects for both general purpose requirements like name-value pair matching to special applications that require the making of complex decisions using scripts. We see a place for a JINI like template filtering presently and other filtering and querying methods can be added as and when required in particular situations.

### 6.2 Handling generic messages

XMessages considers all messages to be just a valid XML structure. So it parses the message to verify the correctness of the XML but does not try to interpret it. To do so, it would have to be aware of the

XML schema and this can be provided by mapping different message types to the corresponding message object. This makes it necessary to maintain a central repository of mappings accessible to different messaging interfaces. This is not viable and a better alternative is to present a generic XML structure of the message that can be understood by individual implementations of the messaging components. This would provide a standard API to the XML form of the message and ease access to its fields without worrying about parsing the XML. Application writers can use this API to write different message handling routines based on the XML content and at the same time, the core messaging layer can be retained without being aware of the message schema.

### 6.3 Dynamic plugins

The message channel can provide a gamut of value added service and be quite powerful as it can act as a central controller for switching messages between applications. But the ability to add such features to the channel must be made more convenient instead of having to write complete implementations of the message channel for each new feature. We are adopting the popular concept of plugins to allow for adding features to a channel incrementally.

Each channel starts of with a default set of core features. When a message is to be processed, it runs through an optional chain of plugins before using the default handling routine. These plugins can be added or removed on the fly by the application starting the channel. This application acts as a controller of the channel and could accept requests to add or remove plugins.

For example, consider a user who has launched a component that publishes events about its status every few minutes to a channel. The user may initially have a simple listener application on her desktop machine to receive these events from the channel. If she needs to go away from her desk and still continue to monitor the status, say for a job completion or exception event, she could request the channel to load a plugin to forward the particular events to her handheld device or cellular phone. If she gets back to her desk before its occurrence, she could unload that plugin and continue to monitor all events on the desktop. The plugin feature could be put to many such uses.

### 6.4 WSDL specialization

Currently, we have made the XMessages API transport independent (i.e.) in the WSDL specification, we have defined all message interfaces without using protocol specific constructs such as SOAP header. We are however investigating an optimized transport binding for XMessages, such as one that uses SOAP headers to describe message control parameters (like *currentToken*, *nextToken*, *leaseDuration* etc.), with the message being placed in the SOAP body directly.

### 6.5 Security

Exchanging messages in a large grid brings into focus the need to verify the authenticity of the message and check the authorization of those allowed to publish and subscribe for messages. This check may be on an individual basis as in a message source or sink trusting one another or on a larger perspective with the publishers and listeners being authenticated by a common channel they subscribe to. The modalities of such a security mechanism needs to be looked into more closely.

### 6.6 Extending framework to other messaging platforms

One of the advantages we attributed to the XMessages system is the ability to extend it to implement other messaging APIs or even act as bridges between messaging systems operating in a diverse environment. We need to demonstrate this by creating implementations of other messaging specifications, specifically JMS and OGSA, that have APIs similar to XMessages.

## 7 Conclusion

In the above paper, we have expounded a simple and reliable messaging framework for distributed scientific applications.

The use of SOAP/XML makes it flexible enough to interoperate with messaging systems on different platforms. This also makes it possible to extend its use as a grid Web Service, offering a candidate for a messaging system on the grid. One downside of using XML is that the performance of transmitting a message on the wire degrades compared to a binary protocol. But we do not perceive this to be a bottleneck for application level messaging that is primarily concerned with coordinating and monitoring.

The publisher/subscriber interface allows 'pull' and 'push' retrieval of messages and these alternatives gain significance in a distributed environment having firewalls and NATs. Using the agents as intermediaries for the publisher and listener ensures reliable delivery of messages and allows for recovery from most failure conditions. The message channel supports a mechanism for asynchronous message passing on a many to many basis among publishers and listeners. In addition, it allows for persistent storage of messages and versatile means for querying historical events.

We believe that the XMessages framework provides a feature rich yet light weight messaging system for grid based applications.