An XML Based System for Dynamic Message Content Creation, Delivery, and Control

Galip Aydin, Ali Kaplan, Ahmet E. Topcu, Beytullah Yildiz, Marlon Pierce[1], and Geoffrey Fox
Community Grids Lab
Indiana University
Bloomington, IN 47404-3730

Ozgur Balsoy
Computer Science Department
Florida State University
Tallahassee, FL

[1]Email: marpierc@indiana.edu

Abstract

We describe the design and implementation of an XML messaging system for creating, delivering and managing general purpose XML messages. We describe message composition wizards, a multipurpose delivery system implementation, and message access role definitions. This system may be used as the foundation for both human readable messaging (such as newsgroupss and registration systems) as well as event-driven application-to-application systems.

1. Introduction

XML [1] messages can be used to provide a computer platform-, programming language-, and application endpoint-independent way for both synchronous and asynchronous communication to take place. Instead of concentrating on endpoint implementations (some of which may be produced by independent developers), we can develop application message formats and use common wire protocols (such as HTTP [2] and SMTP [3]) to transport these messages. These messages then become events in a general purpose message-oriented middleware system.

There are numerous potential applications for such a system, including a) a newsgroup system that allows users to post messages, which can then be delivered by email, through a generated web page, or both; b) a training registration system that allows students to register for classes and instructors to post classes and course materials for students; c) human-to-application communication, in which a human generated message results (for example) in a web page update; and d) application-to-application messaging, in which (for example) an email message from a queuing system announcing the completion of a job results in a the execution of another service, thus serving as an event system to push through a workflow chain.

We identify the following key constituents of our messaging system design: an XML message composing tool for creating valid, well-formed messages; an architecture and implementation for delivering the messages to the appropriate interested listeners (supporting both push and on-demand pull delivery systems); and a user role/access control system to define various levels of users and their privileges. We describe our implementations of each of these ideas in the following sections.

2. Message Composition with Wizards

The first step in our system is creating the XML schemas [4] that define the particular types of messages we wish to transfer for a particular application. Particular XML messages are instances of these general schemas.

The advantage of the XML messaging format is that it only requires the client application to create a valid, well-formed message, which can then be handed off to the system using well established wire protocols. Thus for example a user can create a message posting by hand with

any kind of text editor. Obviously, though, a composing system that reliably generates the message is desirable.

A "wizard" composer can be used to reliably create messages from a particular schema. After defining the necessary schema for a particular application, these schemas have a natural mapping to both user interface components (HTML form elements, Java Swing components, etc.) and data objects (such as JavaBeans). We must thus generate the code for the user interface and the data model.

The process of mapping an XML schema to classes in a particular object oriented programming language is called data binding. This language in our case is Java, and so we wish to cast a particular XML instance into JavaBeans. An XML schema and its instances correspond directly with Java classes and objects. The Java classes have member data corresponding to the schema's elements and attributes and accessor methods ("getters" and "setters") for the data. Tools such as JAXB [5] implementations and Castor [6] for making the conversion, or marshaling, between XML and JavaBeans are available.

The data object bindings may be automatically generated as described above, but we must also develop user interfaces based on the schema that use these data objects. We have initially implemented wizard user interfaces by hand using JavaServer Pages (JSP) [7], mapping the schema elements to corresponding HTML form elements and including corresponding Java data objects generated by Castor in the page. The next step is to automate the user interface creation. We are currently developing such a general purpose Schema Wizard system. We assume that the content to be generated is based on one or more XML schemas. We provide a set of constraints and directives to schema developers by which they can take control over interface generation. The wizard, then by using built-in mappings from schema elements to HTML form elements, is invoked with schemas to generate user interfaces and necessary source code for data handling and validation. The resulting Web forms are used to interact with users and help generate schema-based and validated XML documents.

The Schema Wizard works by mapping each schema primitive type to a JSP template "nugget" (written in Velocity[8] for scripting) that defines both the user interface and the action. A string element, for example, may be mapped to a text field, an enumeration to radio buttons, and so on. These JSP nuggets can be used to build up displays for more complicated types. The final JSP page for a particular schema is simply the aggregation of all the base JSP nugget types that are needed.

Our Schema Wizard currently works for a subset of the XML schema specification. We are working on extending the subset as well as other implementations of the wizard such as an automated interface generation for Web services defined by WSDL documents.

3. Message Posting

XML messages may be posted either through any email client or through the web-based wizard system. The former assumes the user correctly created an XML message in agreement with the schema for the particular application, while the latter (as described in the previous section) creates correct messages.

The posted message actually consists of more than just the XML message created by the wizard or user. The parts of the final posted message are a) a message header, consisting of the regular email header after being converted to XML; b) the message body; and c) optionally one or more attachments to the posting. Attachments may be binary document files attached to the text message during HTTP upload or email postings. We wrap this entire message (XML posting plus optional email headers and MIME attachments) as a SOAP [10] message with MIME attachments.

Each message is assigned a URI as a unique identifier. This defines a hierarchical, searchable structure for messages: a message can contain child messages (for example, a thread in a news group system). For instance, in a newsgroup application of the messaging system, we can create the reply messages as a child of the message which is replied to. By the URI type unique ID, the attachments can be stored to the unique directory which is related to the unique ID. In this structure, we can also store the

messages as an XML file in a directory structure just as we stored attachments. We generate these URIs in an automatic fashion. Top level message names begin with an XML namespace, followed by a number indicating their place in the sequence of posted messages. A child message starts with the name of its parent, followed by a number indicating its place in the sequence of postings to that particular parent.

The message must still be directed to the correct message topic channel. We do this by including a destination tag in the posted message with a URI corresponding to the appropriate message channel (newsgroup topic, for instance).

4. Message Delivery Architecture

We need in general to support two sorts of message delivery mechanisms: a "push" model that immediately sends out the message to the subscribing application and a "pull" model that archives posting that can then be recovered on demand. As we illustrate in Figure 1, we use email message delivery as our push system and a database querying system with a browser front end for pull.

As shown in Figure 1, a user may post messages through an email client. This message is received by the Email Handler, which assigns it a unique URI through the ID Generator service and then publishes it to a message distribution hub. Browser wizard postings work similarly.

We use the Java Message Service (JMS) [9] as our message publishing and subscribing hub. A messaging system will be typically divided into several message channels, such as newsgroup topics or classes in a training system. As shown in Figure 1, we only have one publisher per protocol (message posting wizard for HTTP, for example) and one subscriber per protocol (Email Distributor for SMTP, for example). These are decoupled, so email postings can be later read through a browser client to the archive. We thus are funneling actual end user publishers and subscribers through a small number of publishers and subscriber proxies. The Email Handler, for example, receives all email postings and is responsible for delivering the posted message to the correct JMS message channel. Thus access control rights are enforced by the publisher and subscriber proxies, which consult an access control service (currently implemented in a database, as shown in Figure 1). We describe message access control right definitions in more detail in a later section.

Once the message is posted, it will be both immediately sent out and also archived. Immediate notification ("push") is handled by the Email Distributor. The distributor acts as a subscriber to the JMS publishing hub, and contacts the database to get a list of end subscribers that need immediate notification for postings to a particular message channel.

The archival middleware of the messaging system is responsible for recording the messages and providing (feeding) them to the requesters. These two parts of the system are designed to be independent from the message creation part and the message requester part. For that reason, the server should provide us a functionality that the receiver does not need to know anything about the sender. However, the receiver and publisher have to know the message format. The server should also deliver a message to a client only once. JMS provides this functionality for us.

The messages are received by the Java Messaging Service in the middleware of messaging system. The JMS server can be used to communicate events between Java services running on different hosts. These services, such as Email Distributor, may act as bridges to general purpose protocols. The other modules of messaging systems which generate events register as a publisher to the JMS server. The recorder module of the middleware registers to the JMS server as a subscriber to the publication channels which we want to listen.

Each message is stored to a database to provide persistency. The unique ID is used to search the database. The Message Recorder module completes its mission by storing the message to the database and the attachment to the directory system.

The Message Feeder is completely independent from the recorder part. The requests are received via HTTP GET/POST requests through JSP pages. These requests invoke the feeder to retrieve a message from the database. If the requested information can be found in the

database, an XML file is created dynamically. This file includes the information which the requester asked.

The requester makes two kind of request. One is a request which includes the information of the all messages. The response is in RDF Site Summary [11] format. This RSS file includes information such as the link to the original message, the sender name, and the date. The other request is for a specific message, such as a course or a newsgroup posting. To make this request, the requester takes the RSS file and derives the information to request the message body to obtain the desired information.

The Message Displayer uses RSS URI to construct the e-mail/message hierarchy and to get the body of messages. The Message Feeder constructs the RSS file at the request of the message channel. XSL can be used to extract data from XML based message in order to show the required messages to the users. Message Displayer checks the user's access rights by using the database. User access rights allow users to read from and write into message channel topics.

The confirmed message channel can be used to post or read messages. The interface reads the index structure of the message channel by using an RSS Feeder to get all of the message IDs in order to index the archived messages.

5. User Roles and Access Control

As indicated in previous sections, our message system requires access control rights for security. This also governs the dynamic creation of displays in which users see only the message channels that they are entitled to see.

Users can have several different roles with associated privileges. We present here some specific role-based access levels. We do not address login and authentication here. These are currently assumed to be inherited from some other system using the messaging system. Access controls are based on this external proof of identity.

As shown in Figure 1, our messaging system uses one proxy subscriber per transport protocol and also one proxy publisher per transport protocol. For example, Email Handler is a proxy publisher and Email Distributor is a proxy subscriber. These proxies act as access control points and enforce posting and delivery restrictions using the following roles.

Users are allowed to post and receive messages. Users have privileges to read and optionally write to one or more message channels. They also have additional options with regard to the choice of message delivery mechanism. That is, a user may request message notification by push (email), by pull (through a web interface), or both.

Message Channel Administrators have the authority to assign users access to a specific message channel. An individual may have administration privilege over more than one message channel, and a specific channel has one or more administrators. A channel administrator may also modify the access rights of a user, denying a user the privilege of writing to a particular channel, for example.

Top administrators administer the entire messaging system. In addition to the administrator authorities, this role has the authority to create new messages channels and assign administrators to them.

In the access control system, the top administrator controls the channel administrators, and channel administrators control the channel users. Each channel user has been assigned to a role, and message channels define groups. Members of a channel administrator group can modify, add, and remove user rights from the message channel group by having these control structure. Individuals can have different roles in different groups. For instance, an individual may have only user privileges for one group and administration privileges for another.

We now consider an example use case that illustrates the request and confirmation data flow for the access control structure. For example, Administrator 1 confirms the user request for both User A and User B for the Message Channel l. However, for Message Channel 2, Administrator 2 may only confirm User A's request, while User B's request for Group 2 is rejected. Having all these controls, it is easy to

configure user rights for a message channel, and we have a fine grained access control structure for message channels.

6. Conclusions

We have presented several aspects of an XML messaging system: message composition assistance through schema wizards, a system architecture that supports both "push" and "pull" mechanisms, and access controls on posting and delivery.

Some fundamental development work can be added. As mentioned in the main text, the Schema Wizard is still in development, and we foresee the need to provide additional bindings for user interface widgets in programming languages such as Java.

Authentication and single sign-on is another area that needs to be addressed. We can do this both at the socket/transport level in a mechanism-specific way and at the message level in a more mechanism independent way. Thus we can add (for example) Kerberos support in the JMS communication channels, but we can also verify authentication in the posted messages themselves with message signing.

This work was supported by the US Department of Defense High Performance Computing Modernization Program through the Programming Environment and Training initiative as part of the Online Knowledge Center.

7. References

[1] Extensible Markup Language (XML): http://www.w3c.org/XML.
[2] Hypertext Transport Protocol (HTTP): http://www.w3.org/Protocols/rfc2616/rfc2616.html.
[3] Simple Mail Transfer Protocol (SMTP): http://www.ietf.org/rfc/rfc0821.txt.
[4] XML Schema: http://www.w3.org/XML/Schema
[5] Java Architecture for XML Bindings (JAXB): http://java.sun.com/xml/jaxb/
[6] The Castor Project: http://castor.exolab.org/
[7] JavaServer Pages: http://java.sun.com/products/jsp/
[8] Velocity Project Page: http://jakarta.apache.org/velocity/
[9] Java Message Service: http://java.sun.com/products/jms/
[10] Simple Object Access Protocol (SOAP): http://www.w3.org/TR/SOAP/
[11] RDF Site Summary (RSS): http://groups.yahoo.com/group/rss-dev/files/namespace.html
[12] Balsoy, O. et al. "The Online Knowledge Center: Building a Component Based Portal." Accepted for publication in proceedings of the 2002 International Conference on Information and Knowledge Engineering.

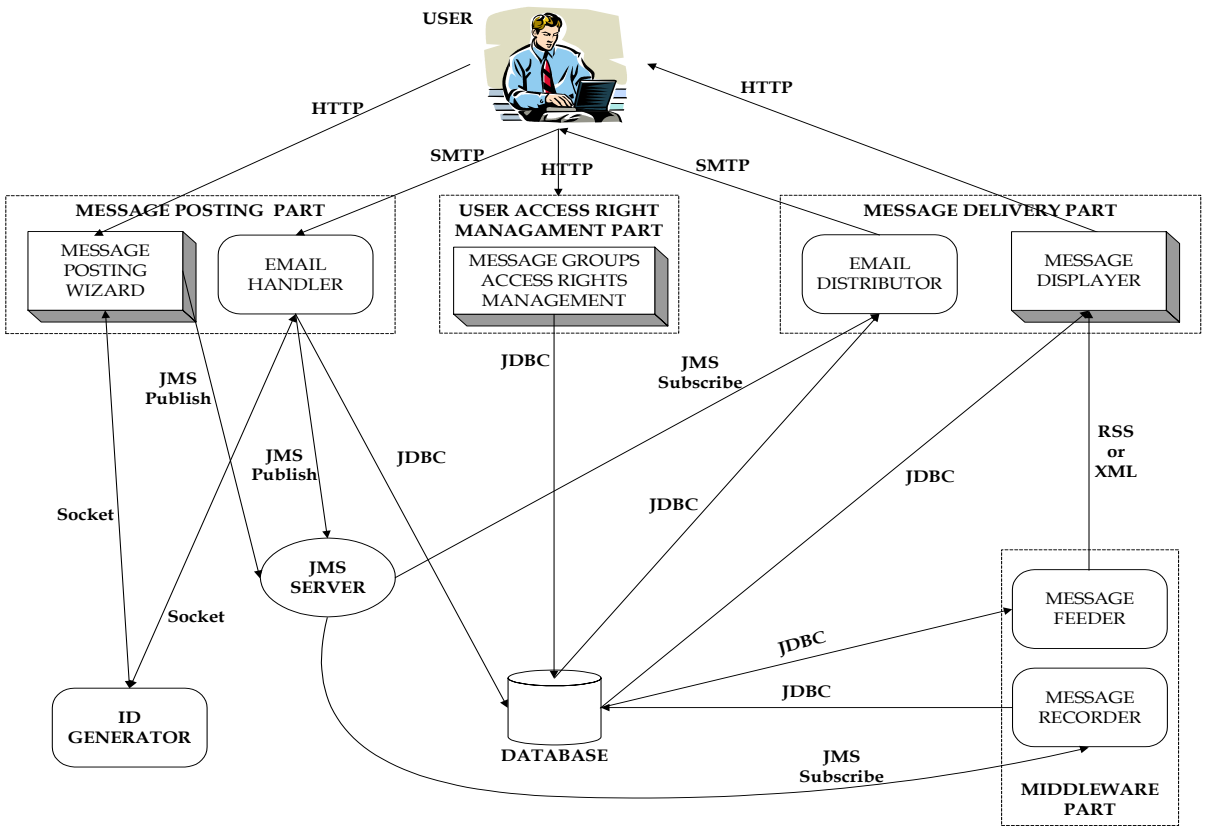

Figure 1 Our XML messaging system architecture supports both immediate delivery and archiving of messages.