FERMILAB-CONF-07-101-CD

A System for Exchanging Control and Status Messages in the NOνA Data Acquisition

Kurt Biery, Glenn Cooper, Stephen Foulkes, Gerald Guglielmo, Luciano Piccoli, Margaret Votava
Fermi National Accelerator Laboratory, Batavia, IL 60510 USA

***Abstract*—In preparation for NOνA, a future neutrino experiment at Fermilab, we are developing a system for passing control and status messages in the data acquisition system. The DAQ system will consist of applications running on approximately 450 nodes. The message passing system will use a publish-subscribe model and will provide support for sending messages and receiving the associated replies. Additional features of the system include a layered architecture with custom APIs tailored to the needs of a DAQ system, the use of an open source messaging system for handling the reliable delivery of messages, the ability to send broadcasts to groups of applications, and APIs in Java, C++, and Python. Our choice for the open source system to deliver messages is EPICS. We will discuss the architecture of the system, our experience with EPICS, and preliminary test results.**

I. Introduction

THE NOνA experiment is designed to study $\nu_\mu \rightarrow \nu_e$ oscillations in the existing Fermilab neutrino beam using detectors on the Fermilab site and in northern Minnesota. Official data taking is expected to start in 2010, and a prototype detector will be built and operated in 2008 with reduced data acquisition needs.

The data acquisition (DAQ) system for the larger detector in Minnesota will contain ~250 data combiner modules (DCMs) and ~180 buffer nodes connected by a Gigabit Ethernet network. Physics data will be collected from front-end digitizer modules by the DCMs and sent to the buffer nodes, from which data in time slices of interest will be sent to mass storage. Control and status messages will use the same network as the physics data, and each DCM and buffer node will send and receive these types of messages as well as transfer physics data. The DAQ system will contain 10-20 control and monitoring applications in addition to the DCMs and buffer nodes, so the total number of nodes that will make use of the messaging system is approximately 450. More details on the NOνA experiment and its data acquisition system can be found in [1] and [2].

II. Message Passing Terminology

To help provide a background for our discussion of the NOνA DAQ messaging system, we provide the following definitions of terms, some of which are commonly used in message passing systems and some of which are unique to the system that we are discussing here:

- message – data in a well-defined format that is used to communicate between distributed applications.
- message passing system (or messaging system) – a means of communication between software components or applications that is loosely coupled. Messages are sent to and read from pre-defined destinations without the sender or receiver needing to know details about the other's location or implementation.
- message producer – a software component that creates messages and sends them to other components in the system.
- message consumer – a software component that receives messages and makes use of the information contained in them.
- message destination – a logical location to which messages are sent and from which they are received.
- point-to-point messaging – a system of message delivery in which each message is delivered to a single consumer. In this model, message destinations may include buffering so that messages may be consumed at a later time than when they were produced.
- publish-subscribe messaging – a system of message delivery in which multiple consumers may receive a particular message. Consumers register interest in a particular destination (subscribe) and subsequently receive one copy of each message sent to that destination. Producers create and send messages to specified destinations (publish) without needing to know the number or location of interested consumers. Destinations do not provide any buffering, so consumers only receive messages after their subscriptions have been registered.
- notification message – an informational message that generates no response from the recipient(s).
- request message – a message that requests that the recipient(s) execute a specific action and report information on the result of that action back to the originator.
- reply message – a message that reports the result of a requested action.

Additional information on messaging terms, concepts, and

patterns can be found in [3] and [4].

III. Requirements

The high-level goals of the message passing system are to provide a robust communications channel between applications at runtime and a straightforward interface for application developers during development.

In addition to these general goals, the system must meet the following specific requirements:

- support several categories of messages – e.g., control commands, status and error reports, and queries for information;
- automatically associate replies with their corresponding requests;
- allow messages to be sent to individual applications ("directed messages") as well as groups of applications ("broadcasts");
- allow applications to specify synchronous or asynchronous receipt of messages;
- make use of a publish-subscribe model for message delivery so that messages may be monitored without disrupting the system;
- support the physical subdivisions of the DAQ network (e.g., into subnets or regions) and the logical subdivision of the DAQ system (e.g., into separate partitions of DAQ elements); and
- provide sufficient data transfer capacity to support approximately 20 request/reply exchanges per second.

It is also desirable to make use of an existing messaging system to provide the low-level message transport and management of subscriptions (a messaging "provider" from our perspective) and allow the replacement of one provider with another with minimal disruption to the full system should that become necessary.

IV. Sample Scenarios

Figures 1, 2, and 3 show several message passing scenarios that we expect in the DAQ system. The first diagram illustrates the sending of notification messages from one application to another. An example of this scenario is the reporting of status information to a central monitor application. The second diagram illustrates the transfer of a request from one application to another and the transfer of a reply back to the original application. An example of this scenario is a control application requesting a state transition from a data taking application, and the data taking application reporting success or failure.

The third diagram illustrates the broadcast of a request to several applications and their corresponding replies. In addition, it shows the presence of a message monitoring application that spies on the outgoing broadcasts for diagnostic purposes. An example of this scenario is a control application sending a transition request to all of the applications in the system, and the applications reporting the success or failure of the transition.

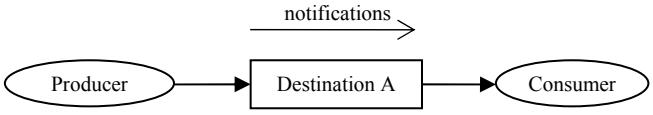

Fig. 1. A simple use case in which messages are sent from one application to another, and no responses are generated.

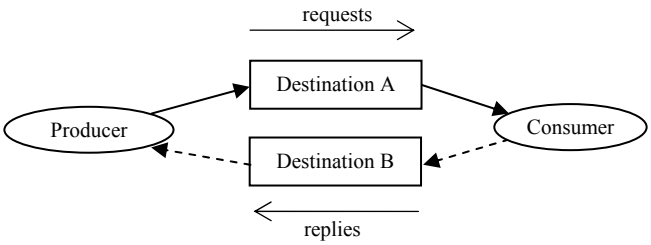

Fig. 2. A typical use case in which requests are sent from one application to another, and replies are returned to the originator. The request messages are shown with solid lines and the replies with dashed lines

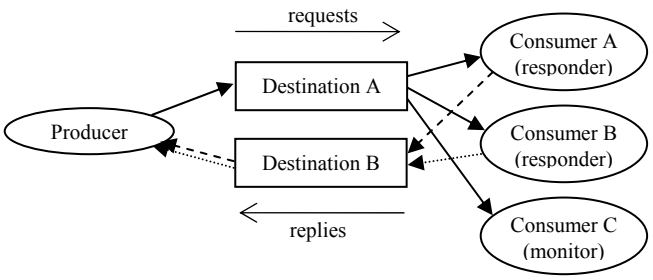

Fig. 3. A typical use case in which messages are broadcast to a group of processes, each member of the group sends a reply to the originator, and the original messages are monitored by a process designed for that task. The request messages are shown with solid lines and the replies from individual consumers with dotted and dashed lines.

Several aspects of these diagrams should be noted. First, in our system, producers and consumers can both send and receive messages. The difference lies in the types of messages that they send and receive. Notifications and requests are sent from producers and received by consumers, whereas replies are sent from consumers and received by producers. As such, the producer and consumer names relate to the production and consumption of the original message, not any associated reply.

A second aspect to note is that although broadcasts in our system simplify the sending of messages from a producer, they don't necessarily simplify the receiving of replies. A producer that broadcasts a request to a group of consumers needs each of the consumers to report the result of the requested action, and it needs to have a list of consumers so

that it can react accordingly if one or more of the consumers do not respond in a timely way. A broadcast of a notification, however, gains the full benefit of grouping the consumers together because replies are not generated for notifications.

V. Architecture and Design

To meet the messaging needs of the NOvA DAQ system, we have created the Responsive Messaging System (RMS). It makes use of a third-party message passing system to provide message transport and management of subscriptions, and it includes two relatively thin layers of software to provide the desired application level interface and encapsulate provider-specific details.

The software stack is shown in Fig. 4. The RMS public layer provides the classes and interfaces to be used by application software. By design, it is independent of any details of a particular underlying messaging provider. The RMS provider layer provides the bridge between the RMS public layer and the third-party system.

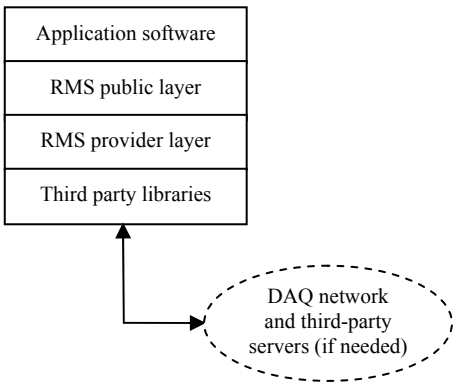

Fig. 4. Outline of the software stack used by the Responsive Messaging System.

A. RMS Public Layer

The primary classes and interfaces contained in the RMS public layer are shown in Figures 5 and 6.

The producer and consumer classes provide methods for sending and receiving messages. Both classes provide internal buffering so that the receipt of messages at the provider level is decoupled from their receipt at the application level independent of whether the application level processing is done synchronously or asynchronously. In addition, the producer class contains logic to filter out uncorrelated replies.

The RmsDestination class provides a provider-neutral representation of a message destination. It is implemented as a set of property name and value pairs, where the property names are predefined keywords that specify the destination. Required properties include "target" and "service". The target property is used to indicate the intended recipient(s), and the service property indicates the category of messages (e.g., control, status, or heartbeat). Additional supported properties include "messageType" (notification, request, or reply) and "partitionNumber" which can be used to specify a subset of the elements in the DAQ system.

The RmsMessage class defines message objects at the public layer, and instances of this class are used for all types of messages (notifications, requests, and replies). Each message contains a header and a body, and the header information includes a unique identifier, the length of time that the message is valid (time-to-live), the timestamp of when the message was sent, the intended destination for the

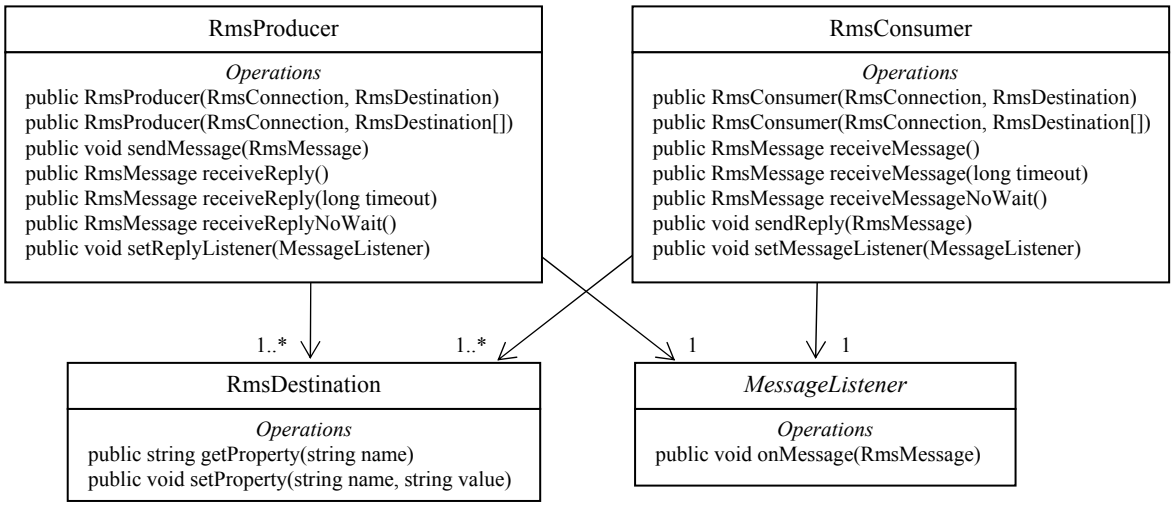

Fig. 5. Partial class diagram for the RMS public layer. Attributes and operations of primary interest are shown. Additional classes and interfaces are shown in Fig. 6.

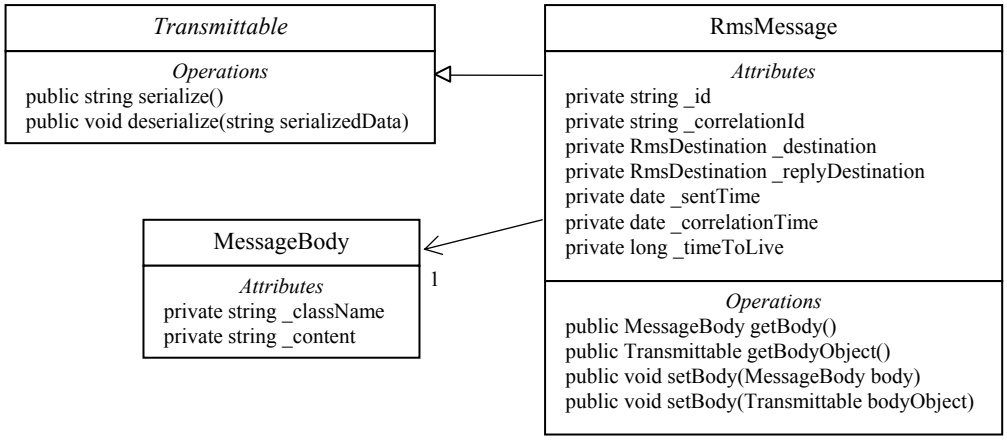

Fig. 6. Partial class diagram for the RMS public layer. Attributes and operations of primary interest are shown. Additional classes and interfaces are shown in Fig. 5.

message, and information about the source of the message. Request message headers additionally include a reply destination, and reply headers additionally include a correlation ID that is used to match the reply to the originating request.

The RmsMessage body is designed to hold the serialization of an application level object. It is implemented as a class that contains the class name of the serialized object as well as the serialized data.

Application level message classes implement an interface (Transmittable) that defines serialize and deserialize methods so that they can be easily stored in and read from RMS message bodies. In addition, the RmsMessage class implements this interface so that producers and consumers can treat these messages in a generic way.

B. *RMS Provider Layer*

The RMS provider layer contains interfaces that are independent of the third-party provider and classes that implement these interfaces for a particular provider. For simplicity, only the provider-neutral interfaces are shown in Fig. 7.

The RmsConnection interface contains the operations that the public layer needs from the provider. These include sending string messages to specified destinations and specifying listeners for receiving messages from particular destinations. The ProviderListener interface is used when the provider layer notifies the public layer that a message has arrived.

VI. The EPICS RMS Provider

Our choice for the initial RMS provider is the Experimental Physics and Industrial Control System (EPICS) [5]. EPICS is an open source control and monitoring system used by many particle accelerators and scientific experiments, and it provides distributed messaging along with many additional features and tools. Communication in EPICS uses a custom protocol named "channel access" (CA), and data reside in well-defined locations called "process variables" (PVs). The system provides server applications that are used to host process variables and provides libraries in C and Java to write data to the process variables and monitor them for changes.

We selected EPICS after evaluating several existing open source messaging systems, including several pure publish-subscribe systems. Although EPICS was not intended to be used as a publish-subscribe messaging system, it was the only candidate that would not have required significant development work to make it production-ready, and it had the advantage of already being used by our group in other projects.

Our use of EPICS is limited to using channel access servers to host process variables to which we write string messages and which we monitor for updates. This effectively creates a publish-subscribe messaging system with the possibility of

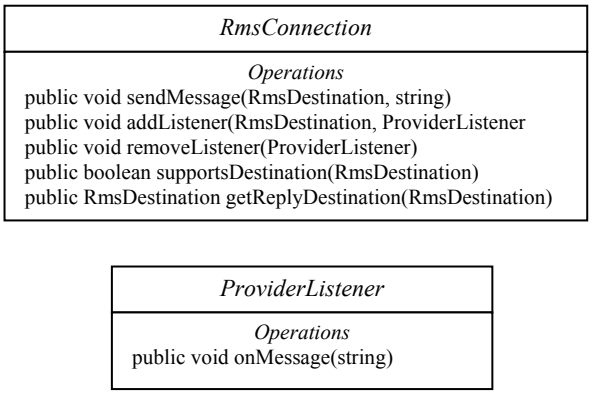

Fig. 7. Class diagram for the interfaces that are part of the RMS provider layer.

multiple applications being notified of an update to a process variable's value. The data format of the process variables used in RMS is a simple byte array of fixed size.

Readers familiar with EPICS will recognize that we are using a small fraction of the functionality and tools available in that system, but our primary needs are for it to handle the low-level message transport and the management of subscriptions.

A. Modifications to the EPICS Base System

Our initial attempts to use EPICS base version 3.14.8.2 for RMS were complicated by the lack of guaranteed delivery of PV updates. Multiple updates to a particular PV in a short period of time occasionally resulted in only the last update being sent to applications that were monitoring that PV. We believe that this is part of the EPICS design, but this behavior did not match our requirements for an RMS provider. An ideal RMS provider has the following behavior:

- attempt to deliver each message to every consumer,
- notify producers of delays in delivering messages, and
- support a configurable timeout for attempts to deliver a message to an unresponsive consumer.

Fortunately, it was straightforward to modify a local copy of the EPICS CA server code base to provide this functionality.

B. Message Delivery with EPICS

As mentioned previously, byte array process variables hosted by channel access servers are the mechanism that we use to pass messages with EPICS. The sending of a message corresponds to writing the serialized message string to the PV, and the receiving of messages is accomplished with callbacks using EPICS monitors.

Within the RMS system, the process variables that are needed for an application to communicate with another application depends primarily on whether the communication will use directed messages or broadcasts. Our model for delivery of directed messages is to define an "inbox" PV for each application and use these inboxes as the destinations for messages and replies. Broadcasts use global outbox PVs for sending messages and global inbox PVs for replies. These two types of communication are illustrated in Figures 8 and 9.

For receiving directed messages, an application creates a consumer associated with a destination that has its target property set to the application name. The EPICS provider layer code takes care of translating the destination object properties to a process variable name. For receiving broadcast messages, an application creates a consumer associated with a destination that has its target property set to the broadcast target. (In both cases, the destination service property and other necessary properties are set appropriately.)

The naming convention for directed message PVs is <targetName>/<serviceName>/inbox. Broadcast process variables are named <regionName>/<partitionNumber>/

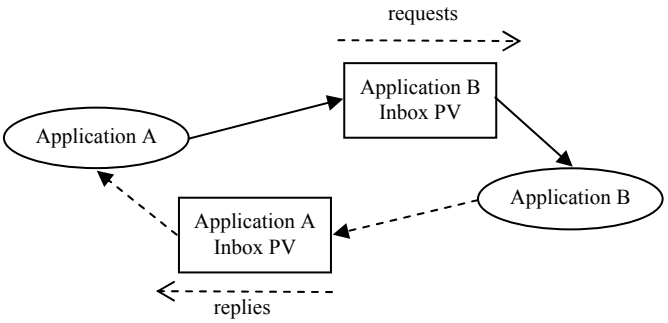

Fig. 8. Directed messages use inboxes associated with individual applications. The inboxes are implemented as EPICS process variables hosted by one or more channel access servers

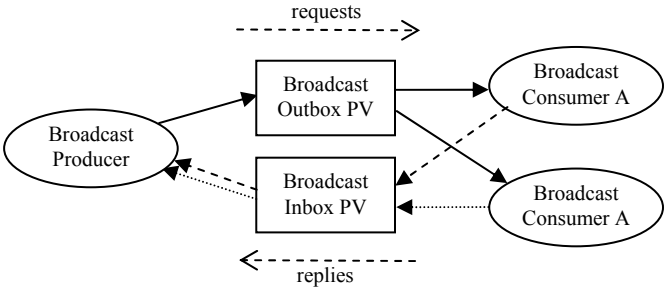

Fig. 9. Broadcast messages use outboxes and inboxes designated for that purpose. The process variables are hosted by one or more channel access servers.

<targetName>/<serviceName>/<mailbox> and include support for multiple regions and partitions.

C. EPICS RMS Provider Layer Implementation Details

The process variables that we use for exchanging messages with EPICS have a fixed size, but we would like to avoid any limitation on the size of messages at the application and public layers. To support this, we developed functionality in the EPICS provider layer to split messages that are longer than the PV byte arrays into fragments when sending messages and to reassemble the fragments when receiving messages.

At the application and RMS public layer levels, directed message and broadcast destinations should be treated equivalently. However, the provider layer needs to know the difference. To support this in the EPICS provider code, we have created a configurable list of broadcast destinations that is used to determine the appropriate translation from generic destination to appropriate PV name.

D. A Suite of EPICS Channel Access Servers

We are currently planning to host a relatively small number of process variables (2-10) on each channel access server. This will cause the overall number of CA servers to be rather

large, but it will help isolate the effect of any problems in one part of the system. In particular, we want to separate the servers that handle different categories of messages (control, error reporting, etc.) from each other so that bottlenecks in delivery of one category don't affect any of the others.

We are currently investigating ways to efficiently deploy and manage the CA servers that will be needed for the full DAQ system.

VII. Alternate Providers

We believe that we have achieved our goal of allowing the underlying messaging provider to be replaced in a straightforward way should that become necessary. Several of the systems that we initially considered as providers used a more traditional publish-subscribe model than what we have created with EPICS. To ensure that these systems continue to be candidates for an RMS provider, we were careful to consider typical publish-subscribe subscription models during the design of RMS.

For reference, the steps needed to add another provider for RMS would consist of implementing the RmsConnection interface for that provider, creating whatever support classes might be needed, and deploying the servers that handle communication for that provider.

VIII. XML Data Binding

Our choice for the format of the serialized message strings is XML. This means that all classes that implement the Transmittable interface must have the ability to translate their internal data into XML and vice-versa.

As an example of a serialized representation of an application level object, the following string shows the XML for an object that requests a state transition:

```
<transitionRequest partitionNumber="0">
  <action>Initialize</action>
</transitionRequest>
```

After this object is wrapped in a full RmsMessage object, its serialization looks like the following string:

```
<rmsMessage
   id="e890ec1b-6993-87bf-1004-487f02533400"
   sentTime="2007-04-23T13:42:33.805-05:00"
   timeToLive="5000">
  <destination>
    <property name="target">
      <value>dcm000</value>
    </property>
    <property name="service">
      <value>control</value>
    </property>
    <property name="messageType">
      <value>request</value>
    </property>
  </destination>
  <reply-destination>
    <property name="target">
      <value>RunControl0</value>
    </property>
    <property name="service">
      <value>control</value>
    </property>
    <property name="messageType">
      <value>reply</value>
    </property>
  </reply-destination>
  <body className="TransitionRequest">
     &lt;transitionRequest
     partitionNumber="0"&gt;
     &lt;action&gt;Initialize&lt;/action&gt;
     &lt;/transitionRequest&gt;
  </body>
</rmsMessage>
```

It should be noted that the XML string from the original transition request is HTML encoded when it is stored in the RmsMessage body.

To facilitate the translation of the Transmittable objects in RMS to XML, we have chosen to use the Castor open source data binding framework for Java [6] and the CodeSynthesis XSD open source data binding compiler for C++ [7]. These tools allow us to specify application level messages in XML Schema Definition files and generate the corresponding classes from the schemas. With a few minor modifications, the generated classes automatically provide the serialization and deserialization functionality that is needed to implement the Transmittable interface.

IX. Sample Usage

As an example of how the RMS APIs will be used, the following list contains the steps that an application would use to send a request and receive the resulting reply:

1. Create an instance of the provider-specific class that implements RmsConnection.
2. Create an RmsDestination object containing the destination to which the message will be sent.
3. Create an RmsProducer instance using the connection and destination objects from steps 1 and 2.
4. If asynchronous processing of replies is desired, create an instance of a class that implements MessageListener and pass that listener to the producer.
5. Create an instance of the desired application level message class that implements the Transmittable interface.
6. Create an instance of the RmsMessage class and add the message class from step 5 to its body.
7. Send the message using the producer sendMessage() method.
8. Read the reply with one of the producer synchronous receive methods or process it in the MessageListener object created in step 4.

The steps to receive a request and send a reply are similar but would use an instance of the RmsConsumer class rather than an RmsProducer.

Applications are not limited to single producer or consumer, nor are they limited to one function (producing or

consuming). A single application may send messages to several destinations using one or more producers and also receive messages from several destinations using one or more consumers.

X. Current Status

At this time, the development of the Java API is nearly complete, the development of the C++ API is ongoing, and the development of the Python API has not yet begun.

Using a small demo system, written in Java, that exercises the broadcasting of transition requests to two simulated DCM applications and receiving the replies, we have measured a sustained rate of approximately 20 exchanges per second. This rate matches the requirement listed in Section III. This test used the Java RMS API and hosted all of the applications needed for the test, including the EPICS channel access servers, on a single Linux node. Higher rates may be achievable with multiple CPUs.

References

[1] NOvA Collaboration. NOvA Proposal and Talks. http://www-nova.fnal.gov.

[2] R. Kwarciany, G.M. Guglielmo, W. Haynes, F.V. Pavlicek, "Nova DAQ: Data Combiner and Timing System," Proceedings of the IEEE Real Time Conference 2007, Batavia, Illinois, May 2007, to be published.

[3] Java Message Service Tutorial. http://java.sun.com/products/jms/tutorial.

[4] G. Hohpe and B. Woolf, *Enterprise Integration Patterns*, Addison-Wesley, 2004.

[5] Experimental Physics and Industrial Control System. http://www.aps.anl.gov/epics.

[6] The Castor Project. http://www.castor.org.

[7] CodeSynthesis XSD, XML Data Binding for C++. http://www.codesynthesis.com/products/xsd.