# CONTROL AND STATE LOGGING FOR THE PHENIX DAQ SYSTEM *

E. Desmond, for the PHENIX Collaboration, BNL, Upton, NY 11973, USA

*Abstract*

The PHENIX DAQ system is managed by a control system responsible for the configuration and monitoring of the PHENIX detector hardware and readout software. At its core, the control system, called Runcontrol, is a process that manages the various components by way of of a distributed architecture using CORBA. Runcontrol is the governor of virtually all aspects of the operation of the online system. We will use a particular component of the distributed system, the messaging system, to showcase several key aspects. The goal of the system is to concentrate all output messages of the distributed processes, which would normally end up in log files or on a terminal, in a central place. The messages may originate from or be received by applications running on any of the multiple platforms which are in use including Linux, Windows, Solaris, and VxWorks. Listener applications allow the DAQ operators to get a comprehensive overview of all messages they are interested in, and also allows scripts or other programs to take automated action in response to certain messages.

Messages are formatted to contain information about the source of the message, the message type, and its severity. Applications written to provide filtering of messages by the DAQ operators by type, severity and source will be presented.

We will discuss the mechanism underlying this system, present examples of the use, and discuss performance and reliability issues.

## OVERVIEW

The purpose of the DAQ messaging system is to achieve maximum available uptime of the PHENIX DAQ system in order to make full use of the available beam supplied by the Relativistic Heavy Ion Collider (RHIC) facility. The DAQ system consists of in excess of 50 distributed processors which are required to configure, monitor and control the DAQ detector components. To effectively control such a system, it is essential that error, process state and component monitoring information be made available to DAQ operators and to DAQ control applications in such a way that actions can be taken on the changing conditions in a timely manner.

The messaging system accomplishes this by providing a message stream into which the control applications can insert error, status, and monitoring messages. This message stream is available to all control and monitoring applications such that the messages can be delivered to DAQ operators or be acted upon by the applications. A major (secondary) goal of the message system was be easily integrated with legacy C++ and Java based DAQ code with minimal code changes. The system system was also to provide integration to a database for message archiving, and to provide an interface to applications which can provide services such as message filtering and which can provide programmatic action if necessary.

These design goals are achieved by basing the message system on CORBA Event Service and by providing an interface to applications code through a custom Streambuf class for C++ applications. These technologies are described next We are using IONA Technologies OrbixAsp 6.1 implementation of CORBA for the Linux platforms, version 5.1 for Solaris and IONA's Orbix/E version 2.1 for the VxWorks platforms.

.

## CORBA EVENT SERVICE

The CORBA Event Service is a standard services defined by the OMG for the decoupled delivery of messages or events from one application to one or more consumer applications. The Event Service is based on a publish / subscribe architecture. It is an architecture where message producers publish messages directly to the event service. Message consumers receive messages by registering an interest with the event service in a particular subject. It is the responsibility of the event service to deliver messages to all registered consumers. The message producers publish their messages by connecting to one or more event channels which are part of the Event Service. Consumer applications, with an interest in receiving messages on a particular subject, register with one or more event channels. The event channels function as independent streams of messages. All messages sent to an event channel are delivered to all registered consumers. Using the event service message producers and consumers are not directly connected to each other, but are loosely connected to each other through an event channel.

There may be many event channels used in the Event Service, each of which is identified by a unique name. Any number of suppliers can issue events to any number

of event channels. Multiple consumers can each subscribe to multiple event channels.

With the event service, messages which are sent between producer and consumer applications are sent asynchronously. Producer applications do not block on the delivery of messages while the consumers receive the messages. The event channel buffers messages while consumers digest them.

The event service operates in either a push or pull model of message delivery. In the push model, event suppliers initiate the transfer of a message by sending the message to the event channel. In the pull model, the message consumer initiates the transfer of a message by requesting the delivery of a message [1].

## *DAQ use of the Event Service*

In the DAQ system, multiple event channels are used to separate the messages from different detector subsystems and to separate control and process state information from monitoring information. This leads to improved message delivery performance and simplified consumer applications as consumers register only with channels whose message content they are interested in. Operators can monitor control state transitions or error conditions by opening message stream monitoring applications at the appropriate channel of interest. This also simplifies the consumer applications as they do not have to parse through the message content for messages of interest or importance to them.

## *Advantages of using the Event Service*

By basing the message system on the CORBA Event Service, producer applications have no direct knowledge of the consumers applications which receive the messages. Thus no code changes are required by the producer application to issue events to the consumers. Messages are only sent to the event channels. Consumer applications may come and go as they wish with no effect on the performance or delivery of messages on the producers part.

By using the Event Service for message delivery, network bandwidth and processor cpu cycles are not wasted polling the applications for their status. The callback method could also be used for control application update of a state change in a server. However, the use of the callback mechanism requires that server applications handle clients which fail to unregister when they are no longer available. In addition, callback mechanisms closely couple the client to the server application [2]. .

## *Cross Platform transparency*

Message producer application in the DAQ system run on Scientific Linux distribution from FermiLab, Sun Solaris 2.6 and 2.8, and Wind River's VxWorks 5.2. Messages are sent and received between applications running on all three platforms. The use of the event service allows the producer and consumer applications to ignore the differences in operating systems, byte order and language implementation. All marshalling and un-marshalling of messages take place transparently to the user application, within the CORBA framework.

## *Push Model Implementation*

The message system, is implemented using the push model of the CORBA event service In the DAQ system, it is the error or process completion operation which is the immediate cause of a message to be initiated. When such a condition occurs, the server then pushes the information into the message stream.

## *Application Interface*

At the time of the development of the message system, there was already in place a large body of largely C++ based server code. It was desirable for the design of the message system was to provide a simple interface to the application level code such that the message system could be added to this legacy code with minimal code modification. The existing legacy code already contained much embedded diagnostic printout statements. We wanted to ensure that the existing diagnostic messages were uniformly forwarded to the message system automatically. This would ensure that all diagnostic messages would be received without requiring the application developers from going through each line of code to redirect the output to the message system. We have done this by overloading the standard C++ Streambuf class. Messages are sent to the custom Streambuf class through the iostream insertion operator <<. The custom streambuf class provides an overloaded sync function which gets called when the stream buffer gets flushed as a result of the endl osteam operator being called. The sync function takes the text which was inserted into the ostream, formats it into the MsgStruct structure, then pushes the structure into the CORBA event service. User applications associate text with messages of specific severity, type and source identity by creating instances of the custom streambuf class. Constructor parameters of the class identify it type, severity and source. Each message class can be associated with a different named channel. Thus each subsystem of the detector can be associated with its own event channel. An example of a typical message interface is the following:

```
 // create class to overload streambuf and connect to
// the event service
Messages mybuffer(orb,root_poa,channelName,256);

// overloads ostream operator to format message to
// type specified in constructor
msg_control *rcmsg = new msg_control(
        MSG_TYPE_CONTROL,
        MSG_SOURCE_RC,
        MSG_SEV_WARNING," rcserver ");
```

```
Msg_control * errmsg = new msg_control(
        MSG_TYPE_CONTROL, MSG_SOURCE_RC,
        MSG_SEV_ERROR,"runcontrol");
// send warning message
 cout << *rcmsg  << "run control warning" << endl;

// send error message
cout << *errmsg << "error warning " << endl;
```

### *Message Structure*

Messages pushed into the message system are formatted into a message structure. This structure is defined in an IDL file. IDL is the descriptive language which is used by CORBA to define the interfaces and data types which can be marshalled between applications. The structure contains fields to identify the message type, source, severity, source component name, and a text field for error specific message description. The message type identifies whether the message is from a control operation, a debug message or other type. The available message types are listed in Table 1.

Table 1: Message Types

| |
|---|
| MSG_TYPE_ONLINE |
| MSG_TYPE_OFFLINE |
| MSG_TYPE_MONITORING |
| MSG_TYPE_CONTROL |
| MSG_TYPE_CODEDEBUG |
| MSG_TYPE_RUNTIME |
| MSG_TYPE_DEFAULT |

Message source identifies which subsystem the message originated from, while the componentname field identifies a particular computer host, or component that is the source of the message.

The Message structure consists of the following definition:

```
struct MsgStruct {
        long    type;
        long    source;
        long    severity;
        long    reserved1;
        long    reserved2;
        long    reserved3;
        string  componentname;
        string  message;
};
```

## UTILITY APPLICATION

While the message delivery mechanism provides for the delivery of messages, it does not provide the applications which make use of those messages. One very useful application we have developed is the message display utility. This is a Java based application which provides for the display of messages to the detector operators. The application gives the machine operators the ability to filter messages by message type, source or severity. Thus operators can view messages which are selected by their severity, or whether they came from a particular component. This is an important quality of service feature which more will be said of later. This utility has proven to be a key diagnostic tool for the detector operators.

A separate logging utility has been developed whose function is to archive the messages. This application listens on all event channels. It currently logs all messages to an ascii log file. The log file name is identified by date and run number to allow operators to review control operation flow or error trace. This utility is currently being updated to archive the messages into a Postgres database to enable the use of SQL statements to simplify the search for error and status information.

### *Performance*

Measurements were made of the event delivery rate from message suppliers to consumers. The test varied the number of suppliers and consumers as well as the number of channels over which the events were sent. In all cases, the supplier and consumer processes were running on 1 Ghz Pentium 4 PCs running the Fermi Lab release of Scientific Linux. The table shows that the throughput of the event has delivered over 500 messages per second.

Table 2: Event Delivery Performance

| **Suppliers** | **Consumers** | **Event Channels** | **messages / second** |
|---|---|---|---|
| 1 | 1 | 1 | 833 |
| 1 | 4 | 2 | 1012 |
| 2 | 2 | 2 | 594 |
| 20 | 1 | 1 | 833 |
| 20 | 1 | 2 | 869 |
| 20 | 20 | 1 | 743 |
| 20 | 1 | 20 | 754 |
| 200 | 1 | 1 | 833 |
| 200 | 1 | 2 | 800 |

## ISSUES AND UPGRADES

### *Guaranteed delivery*

The CORBA event service does not in itself guarantee delivery of events. Most CORBA vendors provide for buffering of undelivered events within the event service, which is provided with IONA's implementation. While it is certainly possible to lose events, we have not reached the performance capacity of this service in past data taking runs. However, for guaranteed delivery the use of the CORBA Notification service provides such a guarantee. In addition, the CORBA Notification service provides such quality of service features as event persistence, event filtering and priority to events. In the Notification Service, event filtering can be specified at different levels so that only messages of a desired content are delivered. At this time, we implement this feature in user level applications. At the time of the original implementation of the message system, the Notification Service was not available on the VxWorks platform and so could not be used in the previous runs. However, implementations of the CORBA Notification service for the VxWorks platform have since become available.

### *Implementation experience*

In order for VxWorks based applications to create event channels in the Orbix event service implementation, they must be able to get a reference to the Orbix event channel factory class. The VxWorks based applications are built using Orbix/E for its CORBA implementation. This orb implementation was originally developed by Object Oriented Concepts, Inc. for their CORBA implementation product Orbacus. Naturally, this implementation does not know about the scope specification of the IONA EventChannelFactory object which is necessary to get a reference to from the event service. Thus in order to enable applications which run under VxWorks to create channels in the Orbix event service a custom factory class was necessary to resolve this scope issue. We have done this with a custom EventChannel factory class.

## FUTURE DEVELOPMENT

While the architecture of the event service has is appropriate for the needs of the DAQ system, the event service lacks the quality of service features which are present in the CORBA Notification service. While we have yet to encounter any of the limitations of the event service in terms of event and channel persistence, the availability of such features are desirable.

## CONCLUSION

The use of the CORBA Event Service as a bases for the messaging system of the PHENIX DAQ system has proven to be useful and productive. The loosely coupled architecture has reduced code maintenance efforts and reduced development time. The system has proven to be scaleable to meet expanded demands and is an effective tool in system diagnostics.

## REFERENCES


[1] http://www.omg.org/.

[2] Henning and Vinoski,"Advanced CORBA Programming with C++",Addison Wesley,1999,p.930