DECOUPLING CONTROL SYSTEM COMPONENTS USING ASYNCHRONOUS PUBLISH/SUBSCRIBE MIDDLEWARE*

Elliott Wolin, Carl Timmer, D.Abbott, W. Gu, V. Gyurjyan, G.Heyes, E.Jastrzembski, D.Lawrence, B. Moffit
Jefferson Lab, Newport News, VA 23606, U.S.A.

Abstract

A speaker at ICALEPCS 2007 advocated the decoupling of control system components through the use of asynchronous communications. The cMsg package from the Jefferson Lab DAQ group implements true publish/subscribe communications using a narrow interface that meets most of the requirements outlined in the talk mentioned earlier. Decoupled or loosely-coupled communication ensures that changes to one part of a control system have no effect on other parts of the system. Asynchronous communication eliminates needless waits and timeouts. And the flexibility of the subscription space and the ability to transmit arbitrary information allows cMsg to be used for virtually any type of control application, including run control, logging, monitoring, hardware control, alarm systems, etc. In this paper we describe how publish/subscribe works, how it differs from client/server communications, and how asynchronous publish/subscribe communications allows for decoupling. We further describe the cMsg package and its narrow API, how it was designed for simplicity and ease of use, how we use it in control systems at JLab, and how we integrate cMsg with EPICS Channel Access.

INTRODUCTION

At ICALEPCS 2007 Stephen Lewis gave a plenary talk on control system longevity [1] where he advocated "decoupling" (and "decentralization") of control system components. He promoted minimizing the number of protocols used and deliberately creating an intellectual "bottleneck" via the use of a single narrow interface to the underlying communications layer. By "narrow" he meant an interface that is simple and does not allow for too much leeway in how it is used, in the sense that different components could use it in incompatible ways.

In the following we describe the asynchronous publish/subscribe paradigm, show how it differs from the client/server paradigm, and how it satisfies Lewis' requirements. We then describe a particular implementation, the cMsg package from the JLab DAQ group. This is followed by a few examples of how we use cMsg at JLab and how it allows for decoupling in our control systems. We end with a summary and conclusions.

ASYNCHRONOUS PUBLISH/SUBSCRIBE COMMUNICATIONS

The asynchronous publish/subscribe interprocess communication model or paradigm has been widely used in industry for decades, and is seeing more widespread use in the physics community. Note that some client/server systems are described using the words "publish" and "subscribe", but they do not implement a true publish/subscribe model.

Under a basic version of the asynchronous publish/subscribe model, message producers "publish" messages to abstract subjects, which are just arbitrary strings. Any producer can publish to any subject at any time, independent what other producers or consumers are doing. No prior registration of subjects is required, and subjects can be created dynamically, at will. There is no connection or "coupling" of a particular subject to a particular producer process.

This is one of many features which distinguish publish/subscribe from client/server models, where in the latter often only one producer/server is allowed to publish to a particular subject.

Message consumers "subscribe" to subjects, and wildcards are often supported. Consumers have no knowledge of the existence of producers and the subjects they publish to, just as producers have no knowledge of consumers and the subjects they subscribe to. A consumer may subscribe to a subject that no producer ever publishes to, and a producer may publish to a subject that no consumer ever subscribes to.

Message publishing and subscribing is asynchronous in that the producer does not block when a message is published, i.e. it does not have to wait for some consumer process to receive it. Consumers receive messages via an asynchronous callback mechanism, usually running in a separate thread, and do not block when the subscription is made.

Thus producers publish messages at will in a "publish-and-forget" mode, and consumers operate in a "subscribe-and-forget" mode. Note that it is common for a single process to be both a message producer and consumer. Also note that all communications are public, which allows other processes to listen in on communications between processes in a transparent manner.

This ability to asynchronously publish and subscribe independent of the existence of other producers and consumers, and to transparently listen in on communications between processes, are key to implementing the decoupling referred to earlier. System designers can implement basic interprocess communication between a set of processes, then at a later date introduce additional producers and consumers that

implement new functionality, with no disturbance to the original system.

For example, imagine two processes communicating within a control system. At a later date a logger process could be activated that subscribes to the same subjects used by the two processes and logs all the communications between them to disk or database, with no disruption to the original system. This could be done for archive or debug purposes, or to implement some new functionality not imagined when the system was originally designed.

Once again, this ability to add functionality incrementally, with no disruption to existing systems, is key to implementing a decoupled system.

THE JLAB CMSG PACKAGE

The JLab cMsg system implements a somewhat more sophisticated version of the publish/subscribe model described earlier, and also is a framework for unifying disparate interprocess communication packages under a single narrow API. Here we only describe its publish/subscribe capabilities (for a full description of the capabilities of the cMsg package see references [2]-[6]).

Instead of just the subject, in the cMsg package a pair of message fields, subject and type, is used when publishing messages and subscribing to them. In all other respects the type field is treated identically to the subject field.

cMsg messages can hold all common fundamental data types, arrays of these types, as well as cMsg messages and arrays of messages. Thus for example a process that receives many messages could bundle them all up in a single message and ship them off to an archiver process.

Message routing is performed by high-performance background server processes (written in pure Java). TCP, UDP, and multicast are supported. Servers can be grouped together into "clouds" which implement hot server failover and least-hop routing. Appropriate deployment of servers in a cloud can be used to optimize traffic shaping. Finally the routing space can easily be broken up into isolated sub-spaces, if desired.

The cMsg API is available in C, C++ and Java, and can run on many flavours of Linux, Solaris, other flavours of Unix, and VxWorks (currently only the Java version works Windows). Endian conversions are handled automatically except for binary data.

The API is designed to be as simple as possible; no interface definition languages or stub generators are needed. A simple C++ program to send or receive a message takes only a few lines (see ref [2]-[6] for examples).

The cMsg API is narrow in that basic messaging functionality is provided, and all other customizations must be done via conventions in the control system (see Table 1). That is, there is only one type of message and one way to fill, publish, subscribe, and receive messages. The entire underlying transport mechanism could be replaced or modified transparently, with no modifications to user code needed. Indeed many aspects of the internals of the cMsg package have changed over the past five years (we are now on major version 3), but the API has hardly changed at all, and programs written five years ago work fine after recompilation [7].

Table 1. Simplified description of API calls

API Call	Description
connect(UDL, name)	Connect to cMsg system specified by UDL for client "name"
disconnect()	Disconnect from cMsg system
send(msg)	Send message asynchronously
flush(timeout)	Flush messages sent from client
syncSend(msg, timeout)	Send message and wait for server response
sendAndGet(msg, timeout)	Send message and wait for receiving client to send response
subscribe(subject, type, callback)	Subscribe to messages of given subject & type, registering callback for incoming messages
unsubscribe()	Remove subscription
subscribeAndGet(subject, type, timeout)	Subscribe to subject & type and wait for one response
start()	Start receiving messages
stop()	Stop receiving messages
monitor(command)	Synchronous call to request monitoring information

Additional useful synchronous capabilities are provided for convenience. These could be implemented by users or developers using only asynchronous cMsg features, but we found it far simpler to build these capabilities into the base package. Note that responses to synchronous calls are private, and that this is the only non-public communication mechanism in cMsg.

Monitoring capabilities exist to identify all servers in a broadcast domain, list all clients for each server, how many messages they have published, which subject/type combinations they are subscribed to, and how many messages have been received for each subscription.

The cMsg package is available on the JLab FTP site [8].

EXAMPLES

The cMsg package is used as a bridge to different or legacy communication systems, as the foundation of the JLab DAQ system run control system, for data transport in low to moderate speed test DAQ systems, to implement an agent-based experiment control system, to transport ROOT from event analysis processes to central display processes, and many other applications. Below we describe a few of these many applications to illustrate the power of cMsg and the utility of decoupling.

cMsgCAGateway

Gateways provide bridges from the cMsg world to different or legacy communication systems. An example is cross-communication between cMsg and EPICS Channel Access (CA) via the cMsgCAGateway. Here a cMsg message sent to the gateway is converted to a CA put, a synchronous call is used to implement a CA get, and a subscription request is converted to a CA monitorOn.

cMsgCommand Utility

The JLab DAQ run control system uses cMsg to transport commands from a central run control server system to individual DAQ components. The run control system relies on XML configuration files to tell it which components are part of a particular DAQ session. Often it is useful to test an individual component in isolation without having to create a special configuration file for the test.

The cMsgCommand utility creates and publishes a cMsg message based on command-line arguments. Using cMsgCommand, at the command line one can simulate the actions of the full run control server facility, but only exercise the component under test. In fact, for small systems one can write short scripts that use the cMsgCommand utility to control a complete DAQ system. The essential point is that DAQ components do not know or care who publishes the commands they respond to.

cMsgLogger

The cMsgLogger utility can subscribe to an arbitrary subject/type combination, and when messages are received it prints summary information to the screen or file, or stores the messages in a JDBC-accessible database.

This utility is frequently used for debugging, where it is set to subscribe to subject "*" and type "*" (i.e. subscribe to ALL subject/type combinations in the messaging space), then print out a summary line for each message received. This allows developers to view and debug all communications between multiple components.

Logging to a file is used for archiving purposes, or for debugging a system where the volume of messages is too large to look at on a screen. Logging to a database has many uses (see below). Once again the decoupling of components is key.

cMsgQueue and cMsgFileQueue

Sometimes a message needs to be processed some time after is published, perhaps because the consumer is busy, or perhaps because it is not even running at the time the message is published. In the latter case, in a pure asynchronous publish/subscribe system, such messages would never get processed. This can be addressed through the use of persistent message queues.

The cMsgQueue and cMsgFileQueue programs provide a temporary message storage mechanism implemented via a database or file-based FIFO or queue. They subscribe to a user-specified subject/type combination and then store all messages received in the queue system. At a later time message consumers can send a special synchronous request message to the queue process, which then removes the message from the queue and sends it to the requester.

Any number of queue message consumers can be run simultaneously since they cannot interfere with each other, another benefit of decoupling in the message system.

daLogMsg Browser

Prototype versions of JLab DAQ components printed error, warning and info messages to the screen, the idea being that we would implement a network-based system at a later date. When the time came we simply replaced the print statements with calls to a daLogMsg() method that just placed the components of the print statement into a message and published it to a standard subject/type.

After publication the messages could be logged and displayed by the cMsgLogger, but we soon created a graphical utility that implemented more functionality. In particular, operators needed the ability to filter and sort messages rather than just see all messages in simple time order.

The daLogMsg browser utility subscribes to the special daLogMsg subject/type, then stores received messages in a large circular buffer. Operators can filter messages on various message fields (e.g. severity must be WARN or greater), select messages only from particular producers, and scroll back and forth through the circular buffer. The browser can additionally display timeline plots of numeric fields in the messages (see Fig 1).

A future version will allow browsing backwards in time via scanning of a database of messages stored by the cMsgLogger utility.

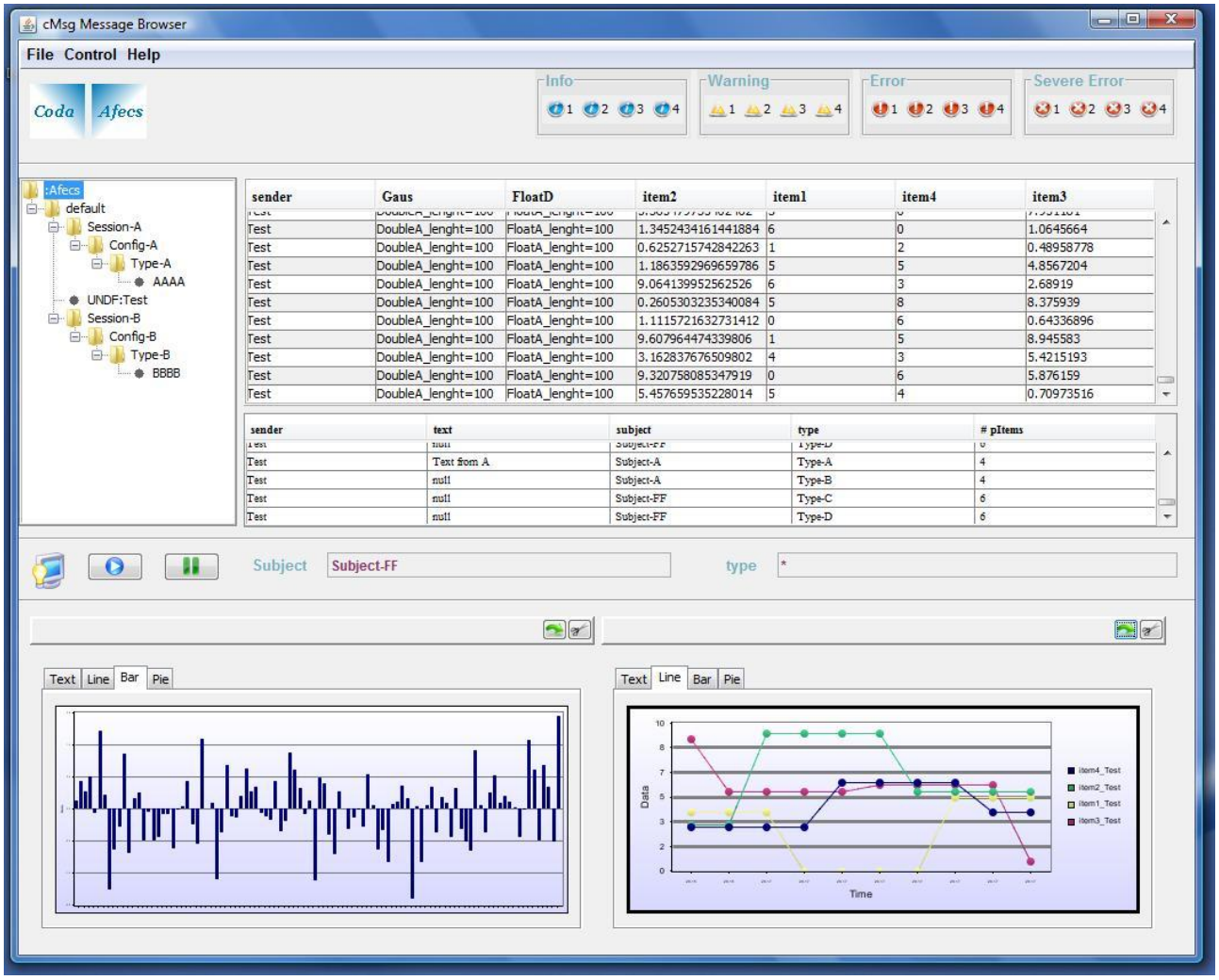

Figure 1: daLogMsg browser display showing list of message senders (upper left), contents of messages received (upper middle), information about messages received (lower middle), histogram of message contents (bottom left) and timeline of message contents (lower right).

RootSpy

Farm-based monitoring and analysis programs often run for long periods of time and use considerable resources. It is extremely useful to be able to monitor the progress of these processes in real-time. In a monitoring situation one might learn that some hardware component is not working properly. In an analysis situation one might learn there is a serious bug in the latest version of the code, and thus want to stop everything before more valuable resources are wasted.

Many of these programs use the CERN ROOT facility to create histograms during the processing. We use the cMsg package to transport ROOT histograms from the farm processes to display GUI's. This was implemented as follows.

A separate thread is run in the analysis process that has access to the ROOT object directories and subscribes to a particular ROOTSPY subject/type. When a "server poll" message is received it responds with assorted identifying information. When a "histogram directory" request is received it responds with a list of all ROOT histograms in its memory. When a "histogram" request message arrives it responds with a message containing a serialized version of a ROOT histogram object, serialized using the TMessage class. This thread runs in parallel with the analysis threads, and since there is no interaction between them, no changes were needed to the analysis program to accommodate the new thread.

When the graphical histogram display program starts it publishes a histogram status poll request message to a standard subject/type to determine what histogram server processes exist. Servers respond via publishing a response to another standard subject/type. Based on the responses (received via a subscription callback) the display utility sends histogram directory requests to selected servers. These respond with their list of histograms, and a graphical directory of available histograms is presented to the user (see Fig. 2).

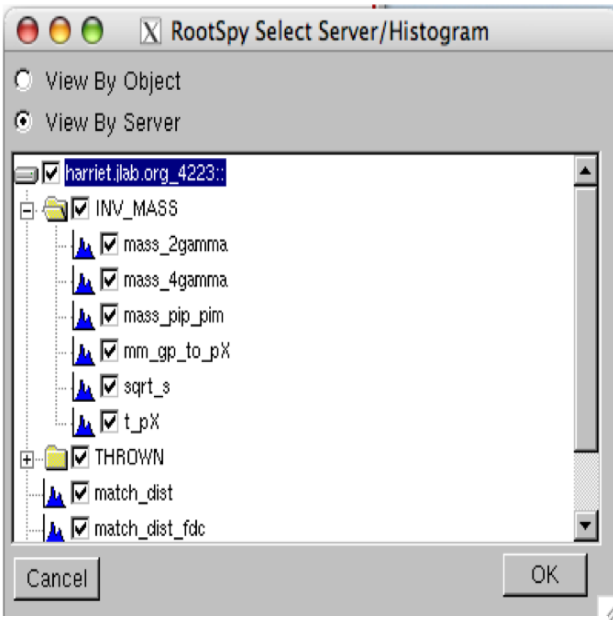

Figure 2: Histogram selection box from ROOTSPY graphical histogram display utility. Shown are lists of histograms provided by histogram server processes. User chooses which histograms to display.

The user chooses which histograms they are interested in and the display program sends requests out for the chosen histograms. The display program then deserializes the ROOT histogram objects contained in the messages when they arrive, and displays the histograms. Note that many independent ROOTSPY display programs can run at the same time.

Once again decoupling allowed us to add the histogram server thread to all analysis programs with no change to the analysis code, and it allows for running multiple independent instances of the ROOTSPY program.

SUMMARY AND CONCLUSIONS

The asynchronous publish/subscribe model is ideal for implementing a decoupled interprocess communication system. Producers can publish messages to abstract subjects with no regard for the activities of other producers or the existence of any message consumers. Consumers can subscribe to subjects with no regard for the activities of other consumers, or to the producers that publish to those subjects. New consumers can be added to implement additional functionality with no change needed to the existing system.

The cMsg package implements a narrow interface that has barely changed over the past five years. It provides basic messaging functionality, and all additional customization is done by developers via conventions in the control system. The result is a very flexible system that remains stable for long periods of time.

In use, we have often created simple systems to implement some basic functionality, then added new functionality via new processes that listen in on the existing messaging and perform new tasks (e.g. archiving or display), with no change to the original system needed. In this way functionality can be built up incrementally and transparently, and modified as needed with no effect on existing systems.

REFERENCES

[1] S. Lewis, “Elements of Control System Longevity,” Proceedings of ICALEPCS 2007, Knoxville, TN, U.S.A.
[2] ????
[3] ????
[4] IEEE Real Time 2006.
[5] CHEP 2007.
[6] PCaPAC 2008.
[7] We recall only one API change, breaking up of a complex call that requested a no-copy transmission of a binary array into two simpler and more natural calls.
[8] ftp://ftp.jlab.org/pub/coda/cMsg.