# Decoupling Control System Components using asynchronous publish/subscribe middleware\*

Elliott Wolin, C.Timmer, D.Abbott, V. Gyurjyan, G.Heyes, E.Jastrzembski, D.Lawrence,
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 client/server systems often describe themselves using the words “publish” and “subscribe”, but they may not implement a true publish/subscribe model.

\*Notice: Authored by Jefferson Science Associates LLC under U.S. DOE Contract No. DE-AC05-06OR23177. The U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce this manuscript for U.S. Government purposes.

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 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 or server is allowed to publish messages to a particular subject (one-to-many vs. many-to-many messaging using a single subject).

Message consumers “subscribe” to subjects, which again are arbitrary strings, 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, and does not have to wait for some 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.

This ability to asynchronously publish and subscribe independent of the existence of other producers and consumers is key to implementing the decoupling referred to earlier. System designers can implement basic interprocess communications between a set of processes, then at a later date transparently add more consumers implementing new functionality, with no disturbance to the original system.

For example, imagine two processes communicating to implement some functionality in 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 messages and arrays of messages. Thus for example a process that received many messages could bundle them all up in a single message and ship them off to an archiver process. Endian conversions are handled automatically.

Message routing is performed by high-performance background server processes (written in pure Java). TCP, UDP, and multicast is 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 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.

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. 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]

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.

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.

Gateways

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). Here a cMsg message sent to a particular subject is converted to a CA put, a synchronous call is used to implement a CA get, and a subscription 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 cMsgMessage 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.

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 a 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. Finally, 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 consumers of the messages in the queue 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.

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

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 stop everything, fix the bug, and not waste any more valuable resources.

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 an information request is received it publishes a message containing a list of all ROOT histograms in memory. When a histogram request message arrives it serialized the ROOT histogram object to a byte array (using the TMessage class) and publishes the histogram to another subject type. 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 histogram display program starts it sends out a ROOTSPY histogram request message that is received by all histogram producers. These all respond with their list of histograms and a graphical directory of available histograms is presented to the user. The user then chooses which histograms they are interested in. The display program then sends out histogram requests for all the chosen histograms, deserializes the ROOT histogram objects when the messages arrive, then 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 publishing thread to all analysis programs with no change to the analysis code, and 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 changed hardly at all over five years. It provides basic messaging functionality, and all additional customization must be done by developers via conventions in the controls system.

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 some new task (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.

IGNORE EVERYTHING PAST THIS POINT EXCEPT THE REFERENCES AT THE END!

|  |
| --- |
| Template-Figure1.png |
| Figure 1: Layout of papers. |

### Title and Author List

The title should use 14 pt bold uppercase letters and be centered on the page. Individual letters may be lowercase to avoid misinterpretation (e.g., mW, MW). To include a funding support statement, put an asterisk after the title and a footnote at the bottom of the first column on page 1; in LaTeX use \thanks.

The names of authors, their organizations/affiliations and mailing addresses should be grouped by affiliation and listed in 12 pt upper and lowercase letters. The name of the submitting or primary author should be first, followed by the co-authors, alphabetically by affiliation.

|  |
| --- |
| Template-Fig2.jpg |
| Figure 2: Example of a full-width figure showing the JACoW Team at their annual meeting in 2008. This figure is labeled with a multi-line caption which has to be justified, rather than centered. |

### Section Headings

Section headings should not be numbered. They should use 12 pt bold uppercase letters and be centered in the column. All section headings should appear directly above the text—there should never be a column break between a heading and the following paragraph.

### Subsection Headings

Subsection headings should not be numbered. They should use 12 pt italic letters and be left aligned in the column. Subsection headings should appear directly above the text—there should never be a column break between a subheading and the following paragraph.

### Paragraph Text

Paragraphs should use 10 pt font and be justified (touch each side) in the column. The beginning of each paragraph should be indented approximately 3 mm (0.13 in). The last line of a paragraph should not be printed by itself at the beginning of a column nor should the first line of a paragraph be printed by itself at the end of a column.

### Figures, Tables and Equations

Place figures and tables as close to their place of mention as possible. Lettering in figures and tables should be large enough to reproduce clearly. Use of non-approved fonts in figures can lead to problems when the files are processed. LaTeX users – please be sure to use non bitmapped versions of Computer Modern fonts in equations (type 1 PostScript fonts are required and their use is described in the JACoW help pages [2]).

All figures and tables must be given sequential numbers (1, 2, 3, etc.) and have captions (10 pt font) placed below figures and above tables being described. Captions that are one line should be centered in the column, while captions that span more than one line should be justified. The LaTeX template uses the ‘booktabs’ package to format the tables

A simple way to introduce figures into a Word document is to place them inside a table which has no borders. This is done in Word as follows:

* Insert a continuous section break.
* Insert two empty lines (will make subsequent editing easier).
* Insert another continuous section break.
* Click between the two section breaks and Format  columns  Single.
* Table  Insert single column, two row table.
* Paste the figure in the first row and adjust the size as appropriate.
* Paste/Type the caption in the second row and apply figure caption style.
* Table  Table properties  Borders and shading  None.
* Table  Table properties  Alignment  Center.
* Table  Table properties  Text wrapping  None.
* Remove the blank lines from in and around the table.
* If necessary play with the cell spacing and other parameters to improve appearance.

If a displayed equation needs a number, place it flush with the right margin of the column (see Eq. 1). The equation itself should be centered, if possible. Units should be written using the roman (standard) font, not the italic font.

  (1)

### References

All bibliographical and web references should be numbered and listed at the end of the paper in a section called “References.” When referring to a reference in the text, place the corresponding reference number in square brackets [3]. A URL may be included as part of a reference, but its hyperlink should NOT be added.

### Footnotes

Footnotes on the title and author lines may be used for acknowledgements, affiliations and e-mail addresses. A nonnumeric sequence of characters (\*, #, †, ‡) should be used. All other footnotes should be included in the reference section and use the normal numeric sequencing.

Word users—do not use Word’s footnote feature (**Insert**, **Footnote**) to insert footnotes, as this will create formatting problems. Instead, insert footnotes manually in a text box at the bottom of the first column with a line at the top of the text box to separate the footnotes from the rest of the paper’s text. The easiest way to do this is to copy the text box from the JACoW template and paste it into your own document. These “pseudo footnotes” in the text box should only appear at the bottom of the first column on the first page.

### Acronyms

Acronyms should be defined the first time they appear.

## Styles

Table 2 summarizes the fonts and spacing used in the styles of a JACoW template (these are implemented in the LaTeX class file).

## Page Numbers

**DO NOT include any page numbers**. They will be added when the final proceedings are produced.

## Templates

Templates and examples can be retrieved through Web browsers like Netscape and Internet Explorer by saving to disk. See your local documentation for details about how to do this.

Template documents for the recommended word processing software are available from the JACoW Website and exist for LaTeX, Microsoft Word (Mac and PC) and OpenOffice for US letter and A4 paper sizes.

Use the correct templates for your paper size and version of Word. Do not transport Microsoft Word documents across platforms, e.g., Mac↔PC.

Please see the help files for instructions on how to install templates in your Microsoft templates folder.

Table 2: Summary of Styles

|  |  |  |  |
| --- | --- | --- | --- |
| **Style** | **Font** | **Space****Before** | **Space****After** |
| Title | 14 ptUppercase except for required lowercase lettersBold | 0 pt | 3 pt |
| Author list | 12 ptUpper- and Lower case | 9 pt | 12 pt |
| Sectionheading | 12 ptUppercasebold | 9 pt | 3 pt |
| Subsectionheading | 12 ptInitial capsItalic | 6 pt | 3 pt |
| Figurecaptions | 10 pt | 3 pt | 6 pt |
| Tablecaptions | 10 pt | 3 pt | 3 pt |
| Equations | 10 pt base font | 12 pt | 12 pt |
| References | 10 pt, justified with0.25” hanging indent | 0 pt | 0 pt |

## Checklist for electronic Publication

* Use only Times or Times New Roman (standard, bold or italic) and Symbol fonts for text—10 pt minimum except References, which can be 9 pt or 10 pt.
* Figures should use Times or Times New Roman (standard, bold or italic) and Symbol fonts when possible—6 pt minimum.
* Check that the postscript file prints correctly.
* Check that there are no page numbers.
* Check that the margins on the printed version are within ±1 mm of the specification.
* LaTeX users can check their margins by invoking the boxit option.

## References

[1] S. Lewis, “Elements of Control System Longevity,” Proceedings of ICALEPCS 2007, Knoxville, TN, U.S.A.

[2] ????

[3] ????

[4] ????

[5] ????

[6] ????

[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.

[9] A.N. Other, “A Very Interesting Paper,” EPAC’96, Sitges, June 1996, MOPCH31, p. 7984 (1996); http://www.JACoW.org.