cMsg – A Publish/Subscribe Interprocess Communication Package and Framework

Elliott Wolin, D.Abbott, V.Gurjyan, G.Heyes, E.Jastrzembski, D.Lawrence. C.Timmer
Jefferson Lab Data Acquisition Group
12000 Jefferson Ave
Newport News, VA 23606 USA

Abstract-**Asynchronous publish/subscribe messaging is a simple but powerful interprocess communication technique that is widely used in industry. It is less widely used in the academic/research world, partly because commercial implementations are fairly expensive.**

cMsg is both a full-featured publish/subscribe package and a framework within which one can deploy multiply underlying communication packages. The underlying packages need not implement asynchronous publish/subscribe messaging. This framework feature allows one to unify all communications under a single, flexible API, and allows for integration of legacy communication packages. Performance is excellent, making cMsg suitable for use in controls applications as well as in high-speed data transfer applications. cMsg will be used extensively by the next generation of experiments at JLab.

I. Introduction

Interprocess communication (IPC) is a vital component of any control or data acquisition system. Many paradigms exist, but asynchronous publish/subscribe (pub/sub) message-passing has been widely adopted for many reasons, particularly by industry. It is extremely general and flexible, efficient implementations exist, and other IPC paradigms can be implemented using it as a foundation. Usage is usually quite simple, and stub generators and interface definition languages are not needed.

The cMsg package includes a complete asynchronous publish/subscribe component that is simple to use “right out of the box.” It supports C, C++, and Java clients, user programming is quite simple, and message transfer is very efficient. The server component is written in pure Java so it runs on any Java Virtual Machine (JVM) independent of the underlying computer architecture. Just a few lines of code are needed to implement simple cMsg IPC, so test programs can be written and working in minutes. With proper firewall configuration cMsg can easily run over distributed networks.

But cMsg is quite a bit more than an asynchronous pub/sub IPC package. It was originally designed for a different purpose, to unify disparate IPC protocols under a single API. Our motivation was our use of numerous legacy IPC packages and protocols in the Jefferson Lab data acquisition package CODA [1]. Replacing these was tedious, as each was being called with its native API. We decided instead to create a package with a single API and a dispatching layer underneath, code everything to the new API, and use a runtime parameter of the API to select which underlying protocol to use in each instance. It was further designed to be simple to add new underlying protocols, both at the client and server levels. This would allow us to replace underlying packages and protocols at will.

Along the way we realized we needed to create new underlying protocols, as existing ones were inadequate to meet the needs of the next generation of experiments at the upgraded JLab 12 GeV accelerator. One JLab experiment (CLAS in Hall B) had extensive experience and great success using a commercial pub/sub package (SmartSockets by Tipco), so we decided that one of the new protocols would implement full pub/sub messaging.

Requirements for the new pub/sub protocol, appropriate for JLab DAQ distributed monitoring and control applications, were:

- Handle moderate message rates (100’s of Hz)
- Handle moderate message sizes (1000’s of bytes)
- Handle hundreds of clients
- Work on Unix (many flavors), vxWorks
- C/C++ and Java API’s
- No commercial components

Note that our emphasis was on robustness, simplicity, and flexibility, and not necessarily on high performance. Very high speed and high volume data transfer, needed in JLab DAQ systems, is implemented via a different, custom-tailored package (the ET system, see report at this conference).

As will be described below, performance of our implementation substantially exceeded requirements, and cMsg pub/sub IPC is now being used for low to moderate rate DAQ systems at JLab and elsewhere.

In Section II we describe the publish/subscribe paradigm. In Section III we discuss using cMsg simply as a pub/sub package.

In Section IV we discuss using cMsg as a framework for implementing multiple underlying protocols and as a proxy server. In Section V we discuss performance of the full pub/sub implementation. In Section VI we discuss the role of Java in real-time and control systems. Finally Section VI contains a summary and conclusions.

II. What is Publish/Subscribe

The asynchronous publish/subscribe interprocess communication paradigm (a type of "Message Oriented Middleware") is widely used in industry and has proven to be very powerful and successful; yet the model is deceptively simple.

In asynchronous pub/sub messaging, producers first fill message objects, then "publish" the messages to abstract "subjects", in a "launch-and-forget" mode. Message consumers "subscribe" to the abstract subjects and provide callbacks to handle messages as they arrive, in a "subscribe-and-forget" mode. Neither producer nor consumer know of each other's existence. A single process can be both a producer and consumer.

The asynchronous nature of the paradigm matches well the asynchronous nature of communication within real-time and online control systems. Here processes are often multi-threaded and perform multiple tasks. Control information arrives sporadically and must be handled as it arrives and on a priority basis. The same applies concerning status information, in that such processes can only send out information when higher priority tasks are not pending.

Note that the flexibility of the subject space allows multiple groups of processes to communicate without interfering with each other.

In contrast, in peer-to-peer messaging, pairs of processes exchange information directly, and every message sent by one of a pair is received by the other. Here the highest throughputs can be achieved since distribution and network overhead can be minimized. However, peer-to-peer messaging does not scale well, as all processes that want to send or receive messages must be connected to each other, and a process that wants to send a message to all other processes must send the message to each individually.

III. cMsg as a Full-Featured Pub/Sub Package

The cMsg package includes a full-featured implementation of the asynchronous pub/sub paradigm with some useful synchronous peer-to-peer mechanisms added for convenience. Unlike other pub/sup packages, cMsg message routing and subscriptions are based on a pair of tags, "subject" and "type" (both are arbitrary strings), and subscriptions support wildcard matching. Message routing is performed behind the scenes by one or more pure Java servers.

Interconnected cMsg servers implement hot-failover such that if one fails its clients automatically will reconnect to a working server. All connections and subscriptions are reestablished, and the only evidence a client might see is a brief delay and possible loss of messages while the system automatically reconfigures.

The cMsg user API is designed to be very simple to use. Below are some code snippets demonstrating how to program common tasks. Note that the UDL (Universal Domain Locator) below is the runtime parameter mentioned earlier used to select the underlying protocol, here the full pub/sub implementation. UDL's are strings and will be fully discussed in the next section. Finally, messages can contain any number of user-settable fields of many types. Below only the text field is used.

Sending a Message

```
#include <cMsg.hxx>

 // connect to cMsg system
 //     UDL selects underlying protocol
 //     name and description are arbitrary strings
 cMsg c(UDL, myName, myDescription);
 c.connect();

 // create and fill message object
 //     subject and type are arbitrary strings
 //     in this example the payload contains a single text field, an arbitrary string
 cMsgMessage msg;
 msg.setSubject(mySubject);
 msg.setType(myType);
 msg.setText(myText);

 // send message
 c.send(msg);
```

Receiving a message

```
#include <cMsg.hxx>

// connect to cMsg system
cMsg c(UDL, myName, myDescription);
 c.connect();

//  subscribe and start receiving
c.subscribe(mySubject, myType, new myCallback(), NULL);
c.start();

     //  do something else…
```

where the callback class is:

```
class myCallback : public cMsgCallback {

   // see user manual for description of userObject
   void callback(cMsgMessage* msg,  void* userObject) {
        cout << "message subject is:   " << msg->getSubject() << endl;
   }
};
```

Synchronous messaging

A synchronous messaging facility is provided for convenience. Here the requester uses a special call to indicate this is a synchronous request. The receiver then marks its response appropriately and it is delivered only to the original requester.

```
#include <time.h>

struct timespec timeout = {1,0};     // one second timeout
cMsgMessage *response = c.sendAndGet(msg,timeout);
   //  exception thrown if no message arrives within timeout
```

(what else needs to be said about the cMsg subdomain???)

cMsg was originally designed to be a thin dispatching layer on top of a number of legacy IPC packages/services/protocols or messaging spaces, with the ability to dynamically add new ones at the client level. We call these messaging spaces “domains”. Domains are specified at runtime, and in general are completely independent. A number of them exist, and additional ones are easily created.

The cMsg domain is a special domain that employs a proxy-server (pure Java) that supports dynamic addition of underlying packages/services/protocols at the proxy-server level. Here the client communicates with the proxy server using a proprietary protocol, and the server performs the IPC request using a specified underlying protocol on the client’s behalf. This solves the problem of clients needing to communicate with a protocol which is unavailable on the architecture they are running on (common on VXWorks). These pluggable protocols in the proxy-server are called subdomains of the cMsg domain.

In a decision we sometimes question, we named the fully-featured pub/sub implementation described in the previous section “the cMsg subdomain of the cMsg domain”. That is, the client connects to the cMsg domain to access the proxy-server, then requests the server to perform the IPC using the pub/sub system described earlier, i.e. within the cMsg subdomain.

Domains (and subdomain information) are specified by a Universal Domain Locator or UDL, a string with syntax similar to and inspired by http URL’s. The UDL is a parameter given to the cMsg connect call, allowing run-time determination of the protocol to use.

Domains need not implement the full cMsg API, and often don’t. Domains can be very simple, e.g. implementing write-only access to a local file (File domain), or complicated, as in the proxy server domain (cMsg domain) described above.

Similarly, many subdomains exist, and they too need not implement all API features. The simplest subdomain is the LogFile subdomain, which implements write-only access to a file by the proxy server. Note that unlike the File domain, where each client writes to its own file, in the LogFile subdomain (of the cMsg domain) many clients can write to the same file.

The most sophisticated subdomain is the cMsg subdomain (of the cMsg domain), which implements the full pub/sub package described in the previous section.

See the User’s Manual [2] for a description of numerous other features of the cMsg package.

V. Performance of pub/sub system

The proxy-server and cMsg subdomain code are written in pure Java (1.5 or later), and server performance is quite impressive. Although little effort was put into optimizing network speed, server performance exceeds our requirements by two orders of magnitude. This has led us to consider uses for cMsg far beyond our original plans, e.g. for low to moderate speed data transfer in smaller DAQ systems.

Below we show measurements of cMsg throughput employing both Java and C clients on Linux and vxWorks. In all cases the cMsg server was running on a 2.4 GHz quad-Opteron RHEL server, and all nodes had Gbit Ethernet interfaces connected to a Cisco Catalyst 4000 series switch.

We identify two regimes: high message rate/small message size, or the “control” regime, and low message rate/large message size, the “DAQ” regime. The former is generally limited by CPU power on the client and server nodes, the latter by network bandwidth and resources required to service the network. The former is best understood from Fig. 1 , the latter from Fig 2.

In Fig 1 we plot message rate vs. message payload size (overhead is 86 bytes) for a number of different conditions. For the top two curves the producer and consumer ran on the same node as the server, so data did not move over a network. In the control regime the server handled over 33,000 messages per second with Java clients, and slightly less for C clients, somewhat surprising since one might expect C client performance to exceed Java client performance. These results place upper limits on server and client performance in our test setup, and are useful when interpreting later results.

For the next two results both producer and consumer were running on separate 2 GHz dual-Xeon RHEL machines. To our surprise again Java clients displayed equal or better performance than C clients over most of the range. In the control regime Java handled over 25,000 messages per second over the network (actually twice, once from producer to server, then again from server to consumer).

The bottom curve is for a vxWorks C producer running on a 1.3 GHz MVME6100 PPC 7457 processor sending messages to a C consumer on a 2 GHz dual-Xeon RHEL machine. In the control regime performance was about the same as for the Linux C producer.

Network bandwidth effects are most clearly seen in Fig 2, where total data throughput is plotted vs. message size, and results become interesting above about 1 kByte message size.

In the non-network case the data transfer rate peaks at about 330 MBytes/sec, but at different payload sizes for Java and C clients. Note that C performance unexpectedly falls off rapidly at large payload size.

In the network case the C rate peaks at about 110 MBytes/sec, or at almost 90% of the full Gbit bandwidth, but then falls off sharply above about 1 MByte payload size, similar to the non-network case. We do not completely understand these falloffs at large payload size, but suspect they may disappear with careful tuning of the C code and network stack parameters.

Java performance peaks at about 80% of the full Gbit bandwidth over a wide range, and does not fall off. We note that in both the C and Java cases the server machine was using an entire CPU to service the network traffic.

vxWorks performance is not nearly as good, not surprising since the CPU and Ethernet hardware are not as powerful as those in the Linux machines, and the vxWorks operating system was not optimized for Gigabit network performance.

VI. Role of Java in Real-Time and Online Systems

Although Java is playing a serious role in many modern DAQ and online systems, it is only commonly used for the least demanding tasks, such as control GUI's. Many people simply do not believe Java is up to more demanding tasks. Our experience and results are quite to the contrary.

We chose to develop the cMsg server and initial client API's in Java because of its many advanced features (esp. in Java 1.5) and the vastly reduced development time, compared to C, we had experienced in other projects. Thus we were able to very quickly modify the Java code as our thinking developed. Once this design/prototype phase was complete we wrote the C client library. This stage took much longer than the previous stage, due to the lack of high-level facilities in C (e.g. concurrent hashmaps) and a number of other issues, even though we were simply duplicating the Java functionality in C (note that the C++ API is simply a wrapper around the C API). The difference was quite striking.

We further had expected that the C performance would exceed Java performance, but this again was not the case. Despite careful tuning of the C code by an experienced C network programmer, and little tuning of the Java code, the Java code out-performed the C code in the majority of our tests. And the fact that the Java code runs at 80% of the Gbit bandwidth demonstrates that there is little left to be gained[1].

VII. Summary and Conclusions

The cMsg system is a simple, powerful, flexible, and open-source implementation of the asynchronous publish/subscribe paradigm, as well as a framework within which one can deploy multiple underlying IPC systems. The underlying IPC systems need not implement pub/sub IPC. cMsg includes support for C/C++ and Java clients, and runs on Unix and vxWorks.

cMsg performance is only limited by network bandwidth on modern processors, and it exceeds our requirements by two orders of magnitude. Thus we now use cMsg in low to moderate rate DAQ systems as well in numerous real-time messaging and control systems.

Note that cMsg use is in no way limited to online systems, and can equally well satisfy a wide variety of interprocess communication requirements. With proper firewall configuration cMsg easily will run on WAN's as well as LAN's.

The use of Java in cMsg greatly reduced our development time compared to C, and Java performance has proven to be excellent, generally exceeding C performance. Our results clearly demonstrate that Java is a serious contender for almost any DAQ or online requirement.

[1] Java performance depends critically on specifying the correct flags to the Java Virtual Machine (JVM). Including the "-server" flag is very important for both client and server. Be sure that the client JVM has plenty of memory via the "-Xms" and "-Xmx" flags. The server garbage collection scheme is also important, and the "-XX:+AggressiveHeap" and "-XX:+UseParrallelGC" flags proved useful.

REFERENCES

[1] "The CEBAF On-line Data Acquisition System", G.W.Heyes et. al, Proceedings of the 1994 CHEP Conference, April 1994.

[2] "cMsg User's Guide", ftp://ftp.jlab.org/pub/coda/cMsg

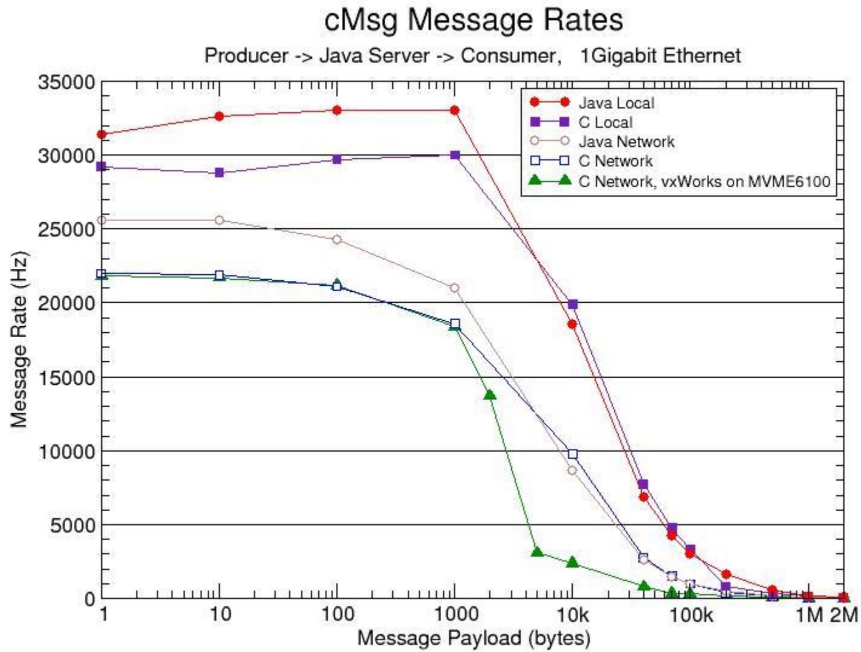

Fig. 1 cMsg message rate versus message payload size.
Overhead is 86 bytes.

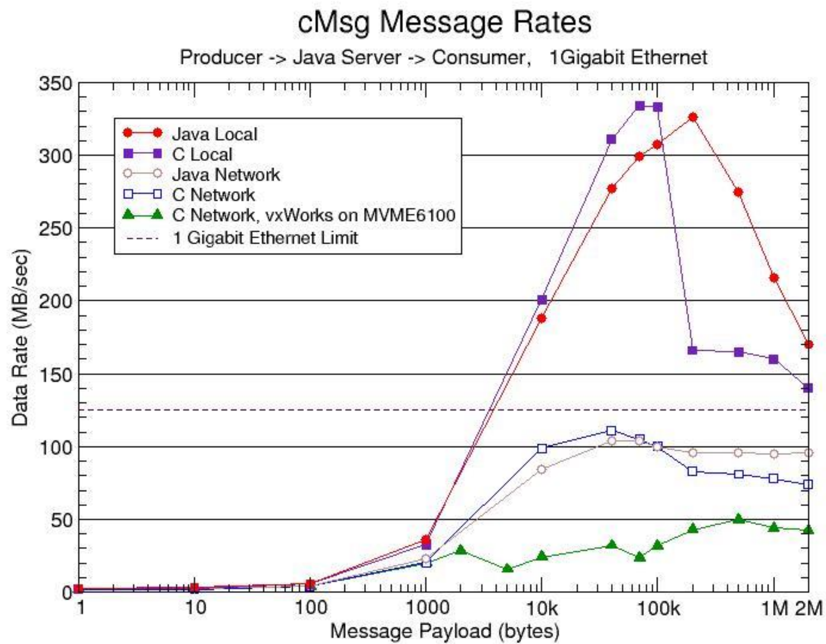

Fig 2. cMsg data transfer rate versus message payload size.
Overhead is 86 bytes.