cMsg – A Publish/Subscribe Package for Real-Time and Online Control Systems

Elliott Wolin, D.Abbott, V.Gurjyan, E.Jastrzembski, D.Lawrence. C.Timmer

***Abstract*--cMsg is a messaging framework designed for use in real-time data acquisition and online controls systems. It provides a single application programming interface (API) to a set of diverse underlying messaging systems. It further implements a proxy system whereby messaging requests are handled by a remote server instead of within the client process. cMsg also includes a built-in full-featured public-domain publish/subscribe messaging system, as well as support for a number of IPC systems commonly used in HENP. Below we first describe the publish/subscribe messaging paradigm and discuss its use in real-time and online systems. Next we describe the philosophy of the cMsg framework and present some details as well as benchmarks using the built-in publish/subscribe messaging system. Finally, since the core of cMsg is written in pure Java, we discuss the suitability of Java for use in real-time and online systems.**

I. Introduction[1]

The CODA [1] data acquisition package at Jefferson Lab (JLab) has been in use and under continual development for almost a decade, and currently employs a number of mutually incompatible interprocess communication (IPC) systems and API's. Recently we decided to unify all interprocess communication under a single API, as well as decrease the number of underlying communication packages used. This should simplify life for developers and users, and allow us to change or add new underlying IPC systems without having to modify application code.

Our API and package requirements are:

- Powerful enough to support publish/subscribe IPC
- Include proxy system
- Handle moderate message rates (100's of Hz) and sizes (100's of bytes)
- Handle hundreds of clients
- Work on Unix (many flavors) and vxWorks
- Provide C/C++ and Java API's
- No commercial components if possible
- Support a number of existing messaging systems
- Must be simple to add additional messaging systems

We required compatibility with pub/sub because it is an excellent match to our needs and because experiments at JLab already use pub/sub IPC. The proxy system is needed because some of our existing IPC systems do not work on all the architectures we must support. The rates, client counts, architectures, and languages match requirements from experiments at the planned JLab 12 GeV upgraded accelerator. We try to avoid commercial components since we distribute software to many groups in and outside JLab, and including commercial components in the past has been problematic and/or expensive. We need to support some of our existing underlying messaging systems for backwards compatibility, although as described below we developed a new pub/sub IPC system as part of this effort. Finally, we expect to need to add new underlying IPC systems in the future.

There are a few packages developed with similar goals. CDEV [2] is also a thin layer on top of multiple underlying IPC systems, but the CDEV API is not powerful enough for our needs, among other problems. Abeans [3] acts as a layer on top of multiple underlying physical control systems, but the Abeans package is designed to solve a different problem than ours.

Note that our emphasis was on robustness, simplicity, and flexibility, and not necessarily on high performance.

[1] Manuscript received June 19, 2005; revised Sep 23, 2005.
Author's address: 12000 Jefferson Lab MS 6B, Newport News, VA, 23606, USA.
Email: wolin@jlab.org.
This work supported by the US Department of Energy.

II. What is Publish/Subscribe

The asynchronous publish/subscribe interprocess communication paradigm 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 knows 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 multiple groups of processes can communicate without interfering with each other via simple subject naming conventions.

In contrast to asynchronous pub/sub communications, in reliable peer-to-peer messaging pairs of client processes exchange information directly, and a channel must be maintained between all process pairs. Here the highest throughputs can be achieved since distribution and network overhead can be minimized. However, peer-to-peer messaging does not scale well with the number of processes as each process must maintain as many open channels as there are processes. In pub/sub systems processes generally maintain a single channel to a server that routes the messages. If multiple servers are used a variety of server routing schemes are possible, including peer-to-peer routing. Even if servers use peer-to-peer routing the system is less complex than the pure peer-to-peer case because there are usually far fewer servers than client processes.

III. What is cMsg

The cMsg package is a framework within which one can deploy multiple underlying IPC systems. It contains a built-in full-featured asynchronous pub/sub system that also includes some useful synchronous peer-to-peer capabilities.

The cMsg package can be used at a number of levels:

1. As an abstract API one can layer on top of an underlying messaging system
2. As a framework for dispatching to multiple underlying messaging systems
3. As a proxy system whereby clients communicate with remote servers that actually connect to the underlying messaging systems
4. As a full-featured stand-alone pub/sub IPC system

Note that the first two levels are primarily of interest to developers (see the cMsg Developer's Guide [4] for details). Most of this report deals with the third and fourth levels (see the cMsg User's Guide [5]).

Note that the pub/sub system included with the cMsg package is complete and full-featured, with the minor addition that cMsg consumers can subscribe to both a message subject and type, and the two are treated equally in terms of message routing and delivery. Also included within the cMsg package are mechanisms to allow communication with EPICS Channel Access, databases, queues, various IPC packages supported by JLab, the commercial product SmartSockets from Tipco, etc, as well as a number of useful auxiliary programs such as message loggers, gateways, command-line utilities, etc. Other IPC packages are being added, including the DIM package from CERN [6].

Messaging Spaces, Domains, and the UDL

cMsg communications are partitioned into messaging spaces called "domains", and a process can connect to multiple domains. Domains are specified via an http-inspired "Universal Domain Locator" or UDL (see [5] for details). Generally messages published within one domain will not be delivered within another domain, although some domains may violate this. If needed, the cMsgGateway can be used to implement generic cross-domain communications.

Domain access is implemented on the client side, and as discussed above a number of domains are supplied in the cMsg package by default. The proxy service mentioned earlier is implemented via the *cMsg Domain.*

cMsg Domain Server and Subdomains

The *cMsg domain* uses a Java server (Java version 1.5 or later) as a proxy or broker for all interprocess communications. Clients communicate with the server using a built-in proprietary protocol, and the (possibly remote) server interacts with the underlying IPC system on the client's behalf. Thus for example, the proxy system allows a vxWorks client to communicate with an IPC system that does not provide a vxWorks API and library. Byte-swapping and other system-dependent transformations are taken care of automatically.

The *cMsg domain* UDL specifies the host and port on which the cMsg domain server is running. The server implements dynamically pluggable subdomains whereby the code that actually performs the messaging can be loaded at runtime (via a subdomain specification in the UDL [5]). .

The full pub/sub system discussed earlier is deployed within the *cMsg subdomain* of the *cMsg domain*. However, there are many other supplied domains and subdomains [5], and developers can add more of each [4]. Also, not all domains and subdomains implement the full pub/sub paradigm, and indeed a number of them implement quite a bit less. For example, the *File domain* simply logs messages to local files.

cMsg Subdomain

The *cMsg subdomain* implements a full-featured asynchronous pub/sub messaging system, with a few synchronous peer-to-peer mechanisms added for convenience. Although commercial pub/sub packages exist that could meet our needs, as mentioned earlier we wanted to avoid commercial packages if possible. Further, none of the public domain packages we knew about included all the features we needed. Thus we decided to attempt implementation our own package in Java. We were surprised how quickly we were able to implement the base functionality, so we decided to continue to develop the full system in Java, C, and C++;

Using the cMsg Package

Below we list some C++ code snippets demonstrating the simplicity and ease of use of the cMsg package:

Sending a Message

```
#include <cMsg.hxx>

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

 // create and fill message object
 cMsgMessage msg;
 msg.setSubject(mySubject);
 msg.setType(myType);
 msg.setText(myText);

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

Receiving a message

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

     //  do something else…
```

where the callback class is:

```
class myCallback:public MsgCallbackAdapter {
   void callback(cMsgMessage msg,
                  void*  userObject) {
       cout << "subject is:   " << msg.getSubject() << endl;
   }
};
```

Synchronous messaging

```
cMsgMessage response = c.sendAndGet(msg,timeout);
  //  exception thrown if no message arrives within timeout
```

Note that the snippets above will compile and run when linked with the standard cMsg libraries, and that no IDL's, stub generators, etc. are needed.

Performance

The cMsg server is written in pure Java, and although little effort was put into optimization, performance exceeds our requirements by two orders of magnitude with a single server and a small number of clients. This has led us to consider uses for cMsg beyond our original plans, e.g. for high-speed data transfer in smaller DAQ systems.

Below we show measurements of cMsg throughput employing both Java and C clients on Linux and vxWorks (we have not yet completed C++ benchmarks). 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.

Future work

We are currently implementing server-to-server communication capabilities in the cMsg domain to allow for load-balancing and other optimizations. We further are implementing an auto-failover feature whereby clients will automatically be connected to another server if their current server dies. Currently clients only get notified when the server dies and must reconnect to another server on their own.

We are installing a multi-node test system that will allow us to run far more clients than is possible on our existing system. When the new system is complete we will be able to test cMsg under more realistic conditions. In particular, we will measure how performance scales as the number of clients increases to a hundred or more, as expected in the next generation of experiments at JLab.

Additional underlying IPC packages will be supported as needed. Currently we are in the process of adding support for DIM [6].

Finally we plan to add extensive system monitoring capabilities to allow clients to get lists of existing servers, clients, subjects, subscriptions, etc.

IV. 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 (especially 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[2].

V. Summary and Conclusions

The cMsg system is simple, powerful, and flexible open-source framework within which one can deploy multiple underlying IPC systems. It includes a built-in full-featured asynchronous publish/subscribe component, support for a number of commonly used IPC systems, as well as a number of useful utilities. It supports C/C++ and Java clients, and runs on Unix and vxWorks.

cMsg performance approaches network bandwidth limits, and generally is only limited by the networking ability of the server machine. Indeed it exceeds our requirements by two orders of magnitude.

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 (although this may change with further tuning of the C components). Our results clearly demonstrate that Java is a serious contender for almost any DAQ or online requirement.

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

[2] CDEV is used by the EPICS community, and can be found at http://www.jlab.org/cdev.

[3] COSYLAB, "Abeans: Application Development Framework for Java", Presented at the ICALEPC conference, Gyeongju, Korea, 2003.

[4] E.Wolin et. al, "cMsg Developer's Guide", ftp://ftp.jlab.org/pub/coda/cMsg

[5] E.Wolin et. al, "cMsg User's Guide", ftp://ftp.jlab.org/pub/coda/cMsg

[6] C.Gaspar et al, "DIM, a Portable, Light Weight Package for Information Publishing, Data Transfer and Inter-process Communication", Proceedings of the CHEP Conference, Padova, Italy, 2000.

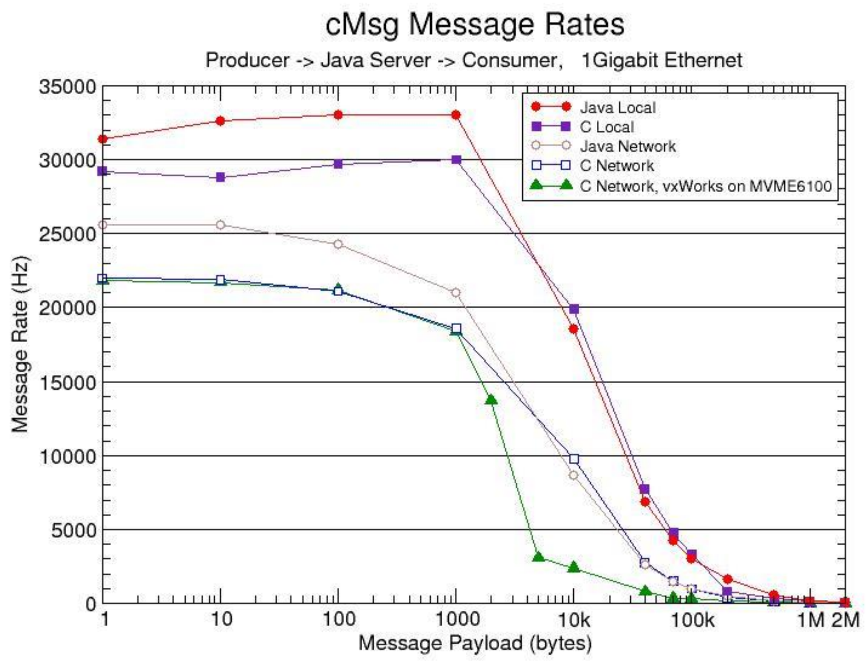

Fig. 1 cMsg message rate versus message payload size.
Overhead is 86 bytes.

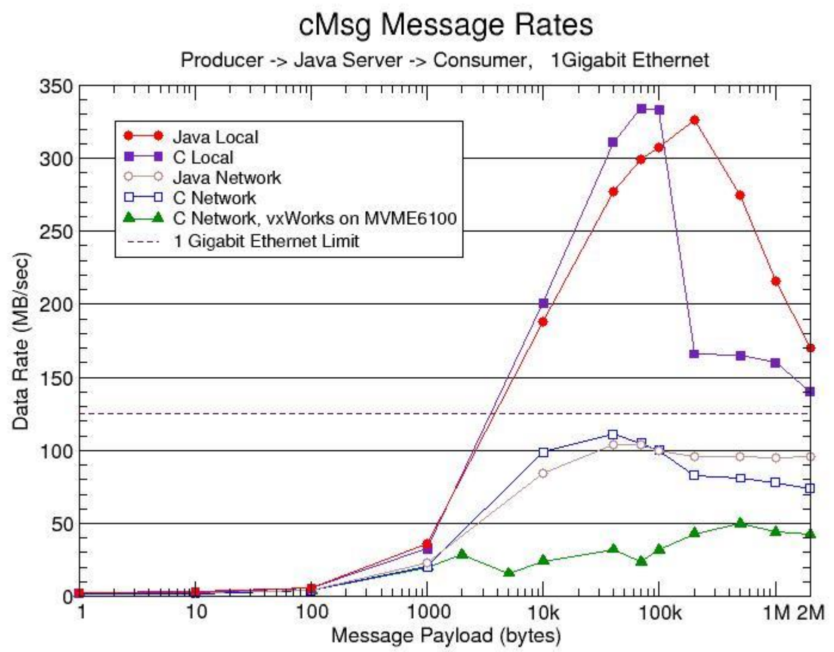

Fig 2. cMsg data transfer rate versus message payload size.
Overhead is 86 bytes.