

Version

3.1

JEFFERSON LAB

Data Acquisition Group

cMsg User's Guide

JEFFERSON LAB DATA ACQUISITION GROUP

cMsg User's Guide

Elliott Wolin
wolin@jlab.org

Carl Timmer
timmer@jlab.org

22-May-2009

© Thomas Jefferson National Accelerator Facility
12000 Jefferson Ave
Newport News, VA 23606
Phone 757.269.7100 • Fax 757.269.6248

Table of Contents

1. Introduction.....	5
1.1. Asynchronous publish/subscribe messaging.....	5
1.2. Synchronous peer-to-peer messaging.....	6
1.3. Implementation.....	6
1.4. Domains	6
2. Messaging Basics.....	7
2.1. Settable fields	7
2.2. Matching with subscriptions.....	7
2.3. XML.....	8
3. Domains and Universal Domain Locators	9
4. cMsg API	10
5. Available Domain Implementations	12
5.1. File Domain.....	12
5.2. CA Domain.....	12
5.3. RC domain.....	13
5.4. cMsg domain	13
5.4.1. Server communication.....	13
5.4.2. Client subscriptions.....	15
6. Starting the cMsg Domain Server	17
6.1. Port Numbers	18
6.2. Subdomains	18
6.3. Passwords.....	18
6.4. Clouds.....	18
6.5. Client throughput regime	19
7. Available Subdomain Implementations	21
7.1. LogFile subdomain.....	21
7.2. CA subdomain	21
7.3. Database subdomain	21
7.4. Queue subdomain.....	22
7.5. FileQueue subdomain.....	22

7.6.	<i>SmartSockets subdomain</i>	22
7.7.	<i>TcpServer subdomain</i>	23
7.8.	<i>cMsg subdomain</i>	23
7.8.1.	<i>Servers</i>	24
7.8.2.	<i>Clients</i>	25
8.	Utilities and Example Programs	27
8.1.	<i>cMsgLogger</i>	27
8.2.	<i>cMsgQueue</i>	27
8.3.	<i>cMsgGateway</i>	28
8.4.	<i>cMsgCAGateway</i>	29
8.5.	<i>cMsgAlarmServer</i>	29
8.6.	<i>cMsgCommand</i>	30
8.7.	<i>cMsgReceive</i>	30
8.8.	<i>Example Programs</i>	31
9.	Client and Server Control and Monitoring	32
9.1.	<i>Control</i>	32
9.2.	<i>Monitoring</i>	32
10.	Java Tutorial	34
10.1.	<i>Connect to a domain</i>	34
10.2.	<i>Create a message</i>	34
10.3.	<i>Send a message</i>	35
10.4.	<i>Subscriptions</i>	35
10.5.	<i>Synchronous methods</i>	36
11.	C Tutorial	38
11.1.	<i>Connect to a domain</i>	38
11.2.	<i>Create a message</i>	38
11.3.	<i>Send a message</i>	39
11.4.	<i>Subscriptions</i>	39
11.5.	<i>Synchronous Routines</i>	40
12.	C++ Tutorial	42
12.1.	<i>Connect to a domain</i>	42
12.2.	<i>Create a message</i>	42
12.3.	<i>Send a message</i>	43
12.4.	<i>Subscriptions</i>	43
12.5.	<i>Synchronous methods</i>	44

A.	Getting, Building, and Installing cMsg	46
1)	<i>Getting cMsg</i>	<i>46</i>
2)	<i>Compiling C/C++ code.....</i>	<i>46</i>
a)	<i>Using SCons.....</i>	<i>46</i>
b)	<i>Using make.....</i>	<i>47</i>
3)	<i>Compiling Java</i>	<i>49</i>
4)	<i>Using C/C++</i>	<i>50</i>
5)	<i>Building Documentation</i>	<i>50</i>
B.	Revision History	51
C.	Contact Information	52

1. Introduction

The cMsg package is designed to provide client programs with a uniform interface to an underlying messaging system via an API powerful enough to encompass asynchronous publish/subscribe and synchronous peer-to-peer messaging. The advantage of using the cMsg API is that client programs need not change if the underlying messaging system is modified or replaced.

But cMsg provides much more than a simple API. The package includes a number of built-in messaging systems, including a complete, stand-alone, asynchronous publish/subscribe and synchronous peer-to-peer messaging system, as well as a persistent network queuing system. Although cMsg is highly customizable and extendable, most users will simply use one of the built-in messaging systems. In addition, a number of useful utilities and examples are provided.

If you are familiar with the publish/subscribe and peer-to-peer paradigms and want to jump right in and learn how to use the cMsg package, skip to the tutorial sections or look at the example programs, and refer back to the next few sections as needed.

The cMsg package can be used at a number of levels, i.e. as an:

- Abstract API to an arbitrary, underlying message system
- Framework for implementation of underlying messaging systems
- Proxy system to connect to remote messaging systems
- Full implementation of publish/subscribe and other messaging systems

The first three levels are primarily of interest to cMsg developers (see the cMsg Developer's Guide). In the rest of this document we describe how to use the built-in messaging systems.

1.1. *Asynchronous publish/subscribe messaging*

The publish/subscribe messaging paradigm is quite powerful, and is commonly used in business and industry. Briefly, in a publish/subscribe system producers publish messages to abstract subjects, and consumers subscribe to subjects they are interested in. A message published to a subject may be delivered to any number of consumers (one-to-many messaging). A client can be both a producer and consumer of messages.

INTRODUCTION

Producers do not know nor care if any clients are subscribed to the subjects they publish to, and consumers neither know nor care about the producers who publish to the subjects they subscribe to.

Note that an additional message field, “type”, plays a similar role as the subject field described above.

1.2. *Synchronous peer-to-peer messaging*

Publish/subscribe messaging is asynchronous in that neither producer nor consumer know about each other’s existence. The cMsg API also includes a number of synchronous messaging mechanisms, in which there is some level of direct communication between the producer and consumer of a message. These range from simple status notifications, which may provide only partial information concerning the status of the producer/consumer transaction, to direct, end-to-end communication between a single producer and a single consumer.

1.3. *Implementation*

cMsg clients can be written in Java (1.5 or higher) on Unix (Linux, Solaris, MacOS X) and Windows, and in C/C++ on Unix and VxWorks. We provide Javadocs and Doxygen docs to document the full Java and C/C++ client API. Note for developers: cMsg domains can be written in C or Java, but subdomains only in Java.

1.4. *Domains*

One term that will be used constantly through this manual is “domain”. In the full implementation of pub/sub and other messaging systems that we are now describing, the cMsg API is actually a top layer which provides multiplexing between different underlying communication systems. Each of these underlying communication systems is referred to as a “domain”. The domain which includes the full pub/sub implementation we call the cMsg domain. Other domains exists which, for example, talk to EPICS channel access or write messages to a file for archiving

2. Messaging Basics

The one thing that ties together all messaging systems under a single umbrella (besides the top layer API) is the type of messages sent between cMsg users. These messages are containers that hold a number of user and system settable fields. The cMsg system itself takes care of setting many fields, including sender, senderHost, senderTime, receiver, receiverHost, receiverTime. In fact, a history can be kept in each message of where it has been.

2.1. *Settable fields*

For the user there are two types of settable fields. The first type of field is limited in number but efficiently handled. This consists of the subject and type fields (strings) which are used to determine who receives the message and also the text (string), byteArray, userInt (32 bit integer), and userTime (64 bit integer) fields. The second type is contained in a “payload” which (though handled less efficiently than the first type) can contain any number of any type of int, float, double, strings, cMsg messages, binary arrays, and arrays of any of these types. This allows tremendous flexibility. After the fields are set, the message can be published. A message can be published many times, and user fields can be modified in between as desired.

Note that the ability for a message to carry other messages as part of its load allows for some interesting possibilities. One such possibility is storing/archiving and restoration of stored/archived messages. Also note that the cMsg system uses the payload to store message metadata. It reserves all payload items whose names begin with the string “cmsg” for itself. For example, the history of a message’s sender name, send time, and send host are stored in the payload fields called cMsgSenderNameHistory, cMsgSenderHostHistory, and cMsgSenderTimeHistory.

There are too many methods/routines that modify messages to list them all here. View the API documentation in either the javadocs or doxygen docs to see the whole list and get an explanation of each.

2.2. *Matching with subscriptions*

MESSAGING BASICS

In all cases, the subject and type fields determine how messages are delivered. Consumers can subscribe to any number of subject/type combinations, and must provide a callback for each (callbacks need not be unique). For those unfamiliar with the term, a callback is a function (C) or an object's method (Java/C++) that is called when a message matching the subscription's subject and type arrives at the client. The message is passed in as a parameter to the function/method.

In the `cMsg` (pub/sub) domain, the wildcard characters “*”, “?”, and “#” are allowed in subscriptions' subject and type, where “*” matches any number of characters, “?” matches a single character, and “#” matches one or no positive integer. Also wildcard constructs like `{i>5 | i=4}` can be used to match a range of positive integers which meet the conditions in the parentheses. The logic symbols `>`, `<`, `=`, `|`, and `&` are allowed along with the letter `i`, any positive integers, and spaces. As an example, a subscription to the subject `abc{i>22 & i<26}` will match a message with the subjects `abc23`, `abc24`, and `abc25`.

The `start()` method enables delivery of messages to the callbacks, and the `stop()` method stops delivery. If delivery is not enabled, messages will be lost to the client. A single message may be delivered to many callbacks in a single client. Note that if a client subscribes to a subject/type and then publishes to the same subject/type, it will receive the message it published.

2.3. XML

A message may be converted into an XML string. In the Java implementation, the various *toString* methods do this and the capability to form a message object from an XML string exists as well. In C/C++ only the ability to convert the message into a string exists. There is some control over whether binary is converted, a compact form is used, or if message metadata is made visible.

3. Domains and Universal Domain Locators

Fundamental to cMsg is the notion of a domain, or messaging space. The cMsg API is actually a top layer which provides multiplexing between different underlying communication systems. Each of these underlying communication systems is referred to as a “domain”. Clients can connect to one or more domains, but subscriptions and messaging activity generally are specific to individual domains. I.e. a publication or subscription in one domain normally has no effect in any other domain. Note that although inter-domain communication in general is not supported, the cMsg gateway utility can bridge domains under certain circumstances. Also, user-written domains may implement inter-domain communication.

Domains are specified via a Universal Domain Locator or UDL. When connecting to a domain, clients supply three pieces of information: the client name, which often needs to be unique (depending on the domain type); a short client description; and the UDL. In analogy with http and other resource locators, the general form of a UDL is:

```
cMsg:domainName://domainInfo?dpar1=val1&dpar2=val2...
```

where the leading cMsg (optional) refers to the cMsg package, domainName identifies the domain, and domainInfo is interpreted by the domain. The optional parameters are domain-specific, and any number of them may be specified. Note that the leading cMsg along with the domainName are NOT case sensitive.

4. cMsg API

There are a handful of methods/functions that do the main work in cMsg. This interface plus the message-manipulating methods are what the user mainly works with. The following table contains a simplified, generic form of the client API (since the Java and C, and C++ versions have slightly different syntax and differ in the number of methods):

Generic cMsg API	
API Call	Description
connect (UDL, description, name)	Connect to cMsg system specified by UDL for client “name”
reconnect ()	Reconnect to cMsg system
disconnect ()	Disconnect from cMsg system
isConnected ()	Is client connected to cMsg system?
send (msg)	Send message asynchronously
flush (timeout)	Flush messages sent from client
syncSend (msg, timeout)	Send message and wait for cMsg system response
sendAndGet (msg, timeout)	Send message and synchronously wait for receiving client to send directed response
subscribe (subject, type, callback, userArg)	Subscribe to messages of given subject & type, registering callback for incoming messages
unsubscribe (sub)	Remove subscription
subscribeAndGet (subject, type, timeout)	Subscribe to subject & type and synchronously wait for one such message
start ()	Start receiving messages
stop ()	Stop receiving messages
monitor (command)	Synchronous call to request monitoring data
shutdownClients (clients)	Command certain clients to shutdown
shutdownServers (servers)	Command certain servers to shutdown
setShutdownHandler (handler)	Install a shutdown handler
subscriptionPause (sub)	Pause message delivery to callback
subscriptionResume (sub)	Resume message delivery to callback
subscriptionQueueClear (sub)	Clear subscription’s message queue
subscriptionQueueCount (sub)	Number of messages in subscription’s queue
subscriptionQueueIsFull (sub)	Is subscription’s message queue full?
subscriptionMessagesTotal (sub)	Number of messages delivered to callback
setUDL (udl)	Reset the UDL for use with reconnect or failover
getCurrentUDL ()	Gets the current UDL

CMSG API

This API contains a number of synchronous functions (`syncSend`, `subscribeAndGet`, `sendAndGet`) simply for the convenience of the user, as each could be constructed from `sends` and `subscribes`. The function `sendAndGet` requires a more detailed explanation. It sends a message in a normal manner while marking it as having been sent by a call to the `sendAndGet` function. A receiver of that message can then construct a special response that will be directed back to the original sender (regardless of its subject and type).

An important detail to keep in mind is that this API may function differently depending on the particulars of each domain's implementation. For example, in one domain "flush" can mean that messages are kept in a buffer until the flush command is issued. In another, however, flush may do nothing as messages are sent immediately during the "send" command. Of course, simply providing an API is of little practical use which is why `cMsg` contains a single implementation of great flexibility.

5. Available Domain Implementations

A number of domains are implemented by default in the cMsg package. The most important is the cMsg domain, which implements a full publish/subscribe and peer-to-peer messaging system in one of its subdomains (see below). Note that calls to API functions not supported by a given domain return a “Not Implemented” error (or throw an exception). Developers can easily implement additional Java and/or C domains.

5.1. File Domain

The File domain implements simple logging of messages to a local file, and the form of the UDL is:

```
cMsg:File://fileName?textOnly=booleanVal
```

where fileName specifies the name of a file locally accessible to the client, and the file is opened in append mode. By default the entire message is logged to the file as an XML fragment. If textOnly=true is specified only a timestamp and the text field is logged. Only the send() and syncSend() messaging API functions are supported. Client names need not be unique in this domain.

5.2. CA Domain

The CA domain implements a simple interface to EPICS Channel Access, similar to the EZCA package, and the form of the UDL is:

```
cMsg:CA://channelName?addr_list=list
```

where addr_list specifies the UDP broadcast address list. Supported messaging API functions are send(), which implements a CA put; flush(); subscribeAndGet() which gets the channel value one time and the timeout is in milliseconds; subscribe(), which implements CA “monitor on”; and unsubscribe, which implements “monitor off”. Callbacks are executed in their own threads.

AVAILABLE DOMAIN IMPLEMENTATIONS

Currently access is only supported for dbl values, and the dbl resides in the text field as a string. Also, the CA domain is currently only implemented for Java clients. Client names need not be unique in this domain.

5.3. *RC domain*

Run control communication is implemented in this domain. This domain is used when the user wishes to create a CODA component -- a component aware of all the CODA state transitions. In order to be informed of the transitions, the correct subscriptions must be made in order to receive the appropriate messages. A full treatment of this topic is beyond the scope of this little user guide, so anyone interested in doing this should contact Vardan Gyurjyan. This section on the rc domain is included merely to inform users that this capability is available. For completeness, the UDL is of the form:

```
cMsg:rc://<host>:<port>/<expid>?multicastTO=<timeout>&connectTO=<timeout>
```

where 1) port is optional with a default of 45200, 2) host is required and may be "multicast", "localhost" or in dotted decimal form, 3) the experiment id or expid is required and does NOT default to the environmental variable EXPID, 4) multicastTO is the time to wait in seconds before connect returns a timeout when a rc multicast server does not answer, 5) connectTO is the time to wait in seconds before connect returns a timeout while waiting for the rc server to send a concluding connect message.

5.4. *cMsg domain*

5.4.1. *Server communication*

The cMsg domain involves use of a proxy server whereby client requests are not handled in the client itself, but instead are forwarded to a remote cMsg server that actually performs the work on behalf of the client. Communication between client and server is handled transparently so the user does not know that a server is involved. The server passes the client's request on to a subdomain (a handler object existing in the server). A cMsg server has many subdomain types to choose from and more can easily be added by a developer. Each subdomain handles the client requests in a particular way. Currently the cMsg domain has cMsg, Channel Access, database, queue, file queue, log file, SmartSockets, and Tcpserver subdomains which are described in the next section. This arrangement works well on VxWorks. Underlying messaging systems that are not available on VxWorks can be accessed from there via a server running on a remote Unix node.

The general form of the cMsg domain UDL is:

```
cMsg:cMsg://host:port/subdomain/subdomainInfo?tag1=val1&tag2=val2...
```

AVAILABLE DOMAIN IMPLEMENTATIONS

For the host parameter, the user may supply a host name or set it to be “multicast” or “localhost”. If it is set to “localhost”, it resolves to the name of the host running the client. If it is set to “multicast”, then the client does a multicast to find the server using the multicast address of 239.220.0.0. Of course, the user may specify a multicast address in dotted decimal form as well. The port used is the one given, or if nothing is given the defaults for both multicasting and TCP connections are 45000. If multicasting is used, the first cMsg server to respond is chosen. If this is not the desired behavior, just make sure your server is running at a unique multicast address and/or a unique port.

The subdomain parameter specifies a particular subdomain implementation, and subdomainInfo is interpreted by the subdomain as are the subdomain parameters. Some of the peculiarities of this UDL include being able to leave off the initial “cMsg” and if the subdomain is missing it will default to the cMsg subdomain. There are three tags that can be used in all subdomains:

Tag Name	Description
cmsgpassword	This is the password needed to connect to the server.
multicastTO	This is the timeout in seconds to wait for server responses to multicasting.
regime	It may be “low”, “medium”, or “high” and defaults to “medium”. This refers to the expected data rate from the client. It’s a hint to the server how best to handle this client. The “high” specification is best used by experts. “Low” is for clients sending normal size messages less than 10Hz or so.

Clients can chose to send messages to the server using either TCP or UDP. This is done by setting a field in the message to be sent. If using a C client, the UDP send is roughly twice as fast as the TCP. In Java there is little difference, but see the API for details. The call to flush() does nothing in the cMsg domain.

Clients using the cMsg domain have the capability to failover to another server when the server they are connected to dies. To use this feature simply supply a semicolon separated list of UDLs in place of a single UDL when connecting to the server. If the client cannot connect to the first UDL on the list, the next is tried and so on until a valid connection is made.

If the server should die during the sending or receipt of messages, the software will try to connect to a UDL on the list and continue on. Any subscriptions are propagated to the new server. Of course, a client who has a subscription will potentially miss messages sent during the brief time it is not connected. Take note that any syncSend(), subscribeAndGet() or sendAndGet() calls will NOT failover, only send(), subscribe(), and unsubscribe() will failover.

The semicolon separated list of UDLs should place the preferred UDL(s) first since it will be given higher priority. If a client is connected to a UDL which is not first in the list and

that connection fails, the software will attempt to establish a connection starting with the first UDL on the list and work its way from there (skipping over the UDL that just failed).

A slight variation on the idea of failing over is that of reconnection. Reconnections are possible in the cMsg domain, meaning that all subscriptions (but not the synchronous syncSend, sendAndGet, subscribeAndGet) are maintained through a client's connection to the server being broken and reestablished. For example, if a client only specifies one UDL (and thus will not failover), and the connection to the server has been broken and disconnect has *not* been called, the user can call connect() again (reconnect() if using C) to reestablish the connection.

Similarly, if a client has a good connection, the user can force a reconnection to another server. To keep subscriptions and switch to another server, the user can reset the UDL by calling setUDL() and then simply call connect() again. This attempts to make a connection starting with the first UDL of the new list. If a connection to another server already exists, it will be terminated before the new one is made.

5.4.2. *Client subscriptions*

There are a number of callback issues users should be aware of when using this domain. All subscription callbacks are run in their own thread with their own messages queues. Thus, use thread-safe programming practices when writing callbacks by either not using shared global resources or by protecting them with mutexes. Also each callback gets delivered a copy of the message arriving at the client, so there is no issue of callbacks interfering with each other by sharing messages.

Since it is possible to have different subscriptions using the same callback, thread-safe coding must be used in such circumstances. It is also possible to have a single callback run in multiple threads by proper configuration of a single subscription. In this case, if the message queue for the subscription begins to fill up, additional threads are started to process messages using the same callback. The above warning about writing a thread-safe callback applies here too.

If the user writes a callback which is designed to accept all matching messages serially, it must be able to “keep up” with the rate of messages arriving in its queue. What happens when that's not the case? Well, TCP provides a back pressure mechanism which will eventually stop message producers from producing any more. The exact chain of events is that once the callback queue is full, the receiving client's TCP buffer will fill up. Once that is full, the cMsg server will not be able to send any more messages to that particular client. Thus the threads in the server sending to this client will block. That may mean that queues of messages (waiting to be sent) in the server fill up which, in turn, may mean that some of the server's TCP buffers fill up. And finally, message producing clients may not be able to send to the server any more due to those full TCP buffers. If the producing

AVAILABLE DOMAIN IMPLEMENTATIONS

client is sending messages using UDP, they will simply be dropped (lost). This is a very unlikely scenario but can actually happen.

The way to avoid this problem is to either produce messages at a reasonable rate, make sure the callback does not take an excessive amount of time to run, configure the callback to automatically delete old messages from its queue when the queue fills up, or configure the callback to be run in multiple threads.

There are method/function calls that the user can make to see if a callback's queue is full, see how many messages are in it, or to completely clear the queue. The user may also pause the callback, resume its operation, and find out how many messages in total have been passed to it.

6. Starting the cMsg Domain Server

You must run a cMsg server if you want to connect to the cMsg domain and its subdomains. The server is not needed for the other domains. There are many options some of which require specialized knowledge of how the server(s) work. The cMsg domain server requires Java 1.5 or later and will produce the following output if run “java -server org.jlab.coda.cMsg.cMsgDomain.server.cMsgNameServer -h” :

```
Usage: java [-Dport=<tcp listening port>]
           [-DdomainPort=<domain server listening port>]
           [-Dudp=<udp listening port>]
           [-DsubdomainName=<className>]
           [-Dserver=<hostname:serverport>]
           [-Ddebug=<level>]
           [-Dstandalone]
           [-Dpassword=<password>]
           [-Dcloudpassword=<password>]
           [-DlowRegimeSize=<size>] cMsgNameServer
```

port	is the TCP port this server listens on										
domainPort	is the TCP port this server listens on for connection to domain server										
udp	is the UDP port this server listens on for multicasts										
subdomainName	is the name of a subdomain and className is the name of the java class used to implement the subdomain										
server	punctuation (not colon) or white space separated list of servers in host:port format to connect to in order to gain entry to cloud of servers. First successful connection used. If no connections made, no error indicated.										
debug	debug output level has acceptable values of: <table border="0"> <tbody> <tr> <td>info</td> <td>for full output</td> </tr> <tr> <td>warn</td> <td>for severity of warning or greater</td> </tr> <tr> <td>error</td> <td>for severity of error or greater</td> </tr> <tr> <td>severe</td> <td>for severity of "cannot go on"</td> </tr> <tr> <td>none</td> <td>for no debug output (default)</td> </tr> </tbody> </table>	info	for full output	warn	for severity of warning or greater	error	for severity of error or greater	severe	for severity of "cannot go on"	none	for no debug output (default)
info	for full output										
warn	for severity of warning or greater										
error	for severity of error or greater										
severe	for severity of "cannot go on"										
none	for no debug output (default)										
standalone	means no other servers may connect or vice versa, is incompatible with "server" option										
password	is used to block clients without this password in their UDL's										
cloudpassword	is used to join a password-protected cloud or to allow servers with this password to join this cloud										
lowRegimeSize	for clients of "regime=low" type, this sets the number of clients serviced by a single thread										

6.1. Port Numbers

Starting off with the simplest of options, the default TCP listening port is 45000 but may be set with the “port” option. For a connecting client, be sure to specify that port number in your client’s UDL if you use this option and are not using the default. Similarly the udp listening port (default 45000) can be specified with the “udp” option. Without going into too much detail, this server has another server within. Two connections are made to this interior server, and its TCP listening port may be set with the “domainPort” option (defaults to 45100). Note that multiple cMsg servers on the same node must use different ports for each of these options.

6.2. Subdomains

Developers can easily implement additional subdomains in Java 1.5 or higher. One can use a subdomain that is *not* built in without too much trouble. When starting up the server, add the following option, “-D<subdomainName>=<className>” where subdomainName is the case insensitive name of the subdomain of interest and className is the name of the Java class implementing the subdomain. When the server starts up and the client connects to it, it first checks to see if the subdomain given by the client’s UDL matches the above option’s subdomain name. If there is no such flag, the environmental variable “CMSG_SUBDOMAIN” is checked to see if it contains the name of the Java class implementing the subdomain. If not, it tries to find the subdomain in the list of those that are built in. More on subdomains can be found in the developer’s guide.

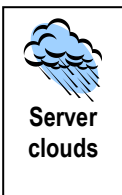
6.3. Passwords

The password option is used not so much as a security measure but simply to avoid confusion by isolating servers from each other. Servers that set the password will allow only clients that specify the password to connect. The end of the client’s UDL must contain the string “?cmsgpassword=<password>” if it’s the first option parameter or the string “&cmsgpassword=<password>” if it’s not the first. The cmsgpassword string is case insensitive.

Some of the options are specific to the cMsg subdomain only. These include -Dserver, -Dstandalone, and -Dcloudpassword. These will be explained in the chapter on the cMsg subdomain.

6.4. Clouds

cMsg servers can be grouped or connected into clouds (note: this has nothing to do with cloud computing). In these clouds, clients of a server in a particular cloud can send and receive messages transparently with clients of other servers in the same cloud. The purpose of this is both to load balance and to distribute computing resources. In principle there is no reason why all clients couldn’t be using the same server. In practice it might be nice to have clients and their server all on the same node for performance reasons. It might also be nice to occasionally talk to other, remote clients. By having servers join



STARTING THE CMSG DOMAIN SERVER

together as part of the same cloud, most of the message traffic can remain local while the occasional message may be sent to or received from a remote client.

To join another cMsg server as part of a cloud, just specify the:

```
-Dserver=host1:port1;host2:port2;hostN:portN
```

option when starting up. The listed host/port combinations may be separated by white space or by any punctuation (except a colon). If white space is used, the whole string will need to be quoted:

```
-Dserver="myhost:33555 yourhost:44555 theirhost:22233"
```

The server being started will try to connect to one of the listed servers starting with the first (left most) and moving to the end (right most). It stops at the first connection made. Once it connects to a single server, it asks that server which other servers it is connected to and connects to those as well. Thus, in a cloud, each server is connected to all of the other servers. These “connections” are nothing more than each server acting as a cMsg client to each of the other servers. I hope that’s not too confusing.

The first server in a cloud may specify a password that it expects joining servers to know. That password is given in the:

```
-Dcloudpassword=<password>
```

option. This feature is not meant for security but only to avoid having extraneous servers join random clouds (i.e. for organization).

If a server does not want other servers to join with him (her?) to form a cloud, it needs to be started with the `-Dstandalone` option.

6.5. Client throughput regime

Last but not least, let me add a word here on the `-DlowRegimeSize` flag. In order to make a server perform better when serving large numbers (100’s to 1000’s) of clients, having clients specifying “regime=low” in their UDLs can greatly help. What this tells the server is that “hey, I’m a client that makes few demands” and so the server diverts fewer resources to that client. For any client that sends and receives normal size messages at no more than 10-20 Hz, it would help to use “regime=low”. Inside the server, a single thread will service several of these low regime clients by use of the “select” statement or at least its Java equivalent. The `-DlowRegimeSize` flag sets how many clients are served by such a single thread. It defaults to 20.

For example, if one server has 3000 such clients and the `lowRegimeSize` is set to 30, then only 100 threads are needed to handle all the clients and everything should work fine. On the other hand, if these clients do not specify “regime=low” in their UDLs, then 3000

STARTING THE CMSG DOMAIN SERVER

threads would be needed on the server to handle them all. This bogs the server down and will cause it to run out of resources such as file descriptors.

7. Available Subdomain Implementations

A number of subdomains are implemented in the cMsg server, including the cMsg subdomain, which implements a complete asynchronous publish/subscribe and synchronous peer-to-peer system (see below).

7.1. *LogFile subdomain*

The LogFile subdomain is similar to the File domain, except that the cMsg server performs the logging. Thus multiple clients can log to the same file, whereas in the File domain the file is unique to the client. Only the send() and syncSend() messaging functions are supported. The general form of the LogFile subdomain UDL is:

```
cMsg:cMsg://host:port/LogFile/filename
```

where fileName is the name of the logging file used by the server. Messages are logged as XML fragments. Client names need not be unique in this subdomain.

7.2. *CA subdomain*

The CA subdomain is similar to the CA domain, but again the cMsg server actually executes the CA library commands, not the client. Only the send(), syncSend(), subscribe(), unsubscribe(), and subscribeAndGet() messaging functions are supported. The general form of the CA subdomain UDL is:

```
cMsg:cMsg://host:port/CA/channelName?addr_list=list
```

where addr_list specifies the UDP broadcast address list. Client names need not be unique in this subdomain.

7.3. *Database subdomain*

In the Database subdomain the cMsg server connects to a database and executes the SQL statement appearing in the text field of a message. Currently SQL queries that return data are not supported (e.g. select). Only the send() and syncSend() messaging functions are supported. The general form of the Database subdomain UDL is:

```
cMsg:cMsg://host:port/Database?driver=myDriver&url=myURL&
```

STARTING THE CMSG DOMAIN SERVER

```
account=myAccount&password=myPassword
```

where myDriver is the JDBC driver to use to connect to the database, myURL is the JDBC URL of the database, and optional myAccount and myPassword may be required by the database. Client names need not be unique in this subdomain.

7.4. Queue subdomain

The Queue subdomain implements network-accessible persistent message queues via a JDBC-accessible database. Clients can post messages to the queue via send() or syncSend(), and retrieve messages from the head of the queue via sendAndGet(). The general form of the queue subdomain UDL is:

```
cMsg:cMsg://host:port/Queue/queueName?driver=myDriver&
url=myURL&account=myAccount&password=myPassword
```

where queueName identifies the queue, myDriver is the JDBC driver to use to connect to the database, myURL is the JDBC URL of the database, and optional myAccount and myPassword may be required by the database.

Note that tables are created in the database by the cMsg server to implement the queue. Also, client names need not be unique in this subdomain.

7.5. FileQueue subdomain

The FileQueue subdomain is identical to the queue subdomain except that the message queue is stored in files rather than in a table in a database. This results in lowered performance, but of course no database is needed. If multiple cMsg servers will access the same FileQueue then the file system on which the queue files reside must support Java 1.5-compatible file locking. Only the send(), syncSend(), and sendAndGet() messaging functions are supported. The general form of the FileQueue subdomain UDL is:

```
cMsg:cMsg://host:port/FileQueue/queueName?dir=myDir
```

where queueName identifies the queue, and optional myDir specifies the directory in which to store the queue files (default is the current working directory of the cMsg server).

Note that many files are created by the cMsg server to implement the queue, so choose the queue directory carefully. Also, a single directory can support multiple file queues. Client names need not be unique in this subdomain.

7.6. SmartSockets subdomain

The SmartSockets subdomain provides a gateway to the SmartSockets (commercial) publish/subscribe interprocess communication package. Only send(), subscribe(), and

STARTING THE CMSG DOMAIN SERVER

unsubscribe() messaging functions are supported. The general form of the SmartSockets subdomain UDL is:

```
cMsg:cMsg://host:port/SmartSockets/projectName
```

where projectName identifies the SmartSockets project.

Note that since in SmartSockets the messaging system is implemented by a “cloud” of interconnected servers, inter-domain messaging is possible if domains connect to SmartSockets servers within the same cloud. Client names must be unique in this subdomain.

7.7. *TcpServer subdomain*

The TcpServer subdomain provides access to tcpsever processes running on VxWorks or Unix processors (tcpserver is part of the CODA data acquisition package at JLab). Only the sendAndGet() messaging function is supported, and communication with the tcpserver is stateless. Commands placed in the request message text field are forwarded to the tcpserver for execution, and the resulting output is returned in the text field of the response message. The general form of the tcpserver subdomain UDL is:

```
cMsg:cMsg://host:port/TcpServer/srvHost:srvPort
```

where srvHost and srvPort refer to the host name and port where the tcpserver process is running. Client names need not be unique in this subdomain.

7.8. *cMsg subdomain*

The cMsg subdomain implements a complete asynchronous publish/subscribe and synchronous peer-to-peer interprocess communication package. All cMsg messaging API functions are supported. Inter-server communication is supported. The general form of the cMsg subdomain UDL is:

```
cMsg:cMsg://host:port/cMsg/namespace/?tag=value&tag2=value2 ...
```

For the host parameter, the user may supply a host name or set it to be “multicast” or “localhost”. If it is set to “localhost”, it resolves to the name of the host running the client. If it is set to “multicast”, then the client does a multicast to find a server on an unknown host using the multicast address of 239.220.0.0. Of course, the user may specify a multicast address in dotted decimal form as well. The port used is the one given, or if nothing is given the defaults for both multicasting and TCP connections are 45000. If multicasting is used, the first cMsg server to respond is chosen. If this is not the desired behavior, just make sure your server is running at a unique multicast address and/or a unique port.

The optional namespace parameter identifies a messaging namespace. A namespace may consist only of ASCII word characters [a-zA-Z0-9_], and if not supplied a default

STARTING THE CMSG DOMAIN SERVER

namespace is used (defaultNamespace). Client names must be unique in the given namespace. Messages do not cross namespaces. Thus, a client who subscribes to a subject/type in a particular namespace will only be able to receive messages from a producing client in that same namespace. There are a number of tags that can be used:

Tag Name	Description
cmmsgpassword	This is the password needed to connect to the server. Valid for all cMsg subdomains
multicastTO	This is the timeout in seconds to wait for server responses to multicasting. Valid for all cMsg subdomains.
regime	It may be “low”, “medium”, or “high” and defaults to “medium”. This refers to the expected data rate from the client. It’s a hint to the server how best to handle this client. The “high” specification is best used by experts. “Low” is for clients sending normal size messages less than 10Hz or so. Valid for all cMsg subdomains.
failover	It may be “cloud”, “cloudonly”, or “any” and defaults to “any”. This specifies the manner in which to failover from one server to another. If set to “any”, failover of client can go to any of the UDLs given in the argument to connect(). If set to “cloud”, failover will go to servers in the cloud first, and if none are available, then go to any of the UDLs given to connect(). If set to “cloudonly”, failover will only go to servers in the cloud. Valid for the cMsg subdomain only. More on this below.
cloud	It may be “local” or “any” and defaults to “any”. This specifies the manner in which to failover to a cloud. If set to “any”, failover of client to a server in the cloud will go to any of the cloud servers. If set to “local”, failover of client to a server in the cloud will go to a local cloud server first before others are considered. Valid for the cMsg subdomain only. More on this below.

Some of the peculiarities of this UDL include being able to leave off the initial “cMsg” and if the subdomain is missing it will default to the cMsg subdomain.

7.8.1. Servers



The cMsg subdomain is the most complex of all the subdomains. Let’s start out by describing the servers and how they connect together (see also section 6.4). cMsg servers may be connected together into what is called a cloud. This cloud is only meaningful to clients which use the cMsg subdomain. A client of one of the servers in a cloud can subscribe to messages produced by any client connected to any server in that cloud within the cMsg subdomain. Similarly a message-producing client can sent to any subscribers in the cloud if it is connected to one of the cloud servers.

To get servers to connect to each other, the “–Dserver=<hostname:serverport>” option must be given to the JVM when starting up all except the first server. In the other words,

STARTING THE CMSG DOMAIN SERVER

the first server is started up normally. A second (connecting) server must be started with the above option giving the host and port of the first server as described. Any subsequent servers joining the cloud must also use the above option giving the host and port of any one of the servers already in the cloud.

For example, let's say that we start up a server on the host "hal" on port 45001:

```
Java -server org.jlab.coda.cMsg.cMsgDomain.server.cMsgNameServer
```

Then the second server would be started like:

```
Java -server -Dserver=hal:45001 org.jlab.coda.cMsg.cMsgDomain.server.cMsgNameServer
```

It may be that the first server does not want any one joining him and forming a cloud. In that case the first server would start up with the flag "-Dstandalone" and all joiners would be rejected.

It is also possible to protect a cloud from having just anyone join. A password can be instituted by having the first server start up with the "-Dcloudpassword=<password>" option. If the password were set to say ... abracadabra, then we would type the following at the command line:

```
Java -server -Dcloudpassword=abracadabra ... cMsgNameServer
```

Any server which wanted to join that server to make a cloud must have the same option given. All others will be refused.

7.8.2. *Clients*

Clients using the cMsg subdomain have the capability to failover to another server when the server they are connected to dies. To use this feature simply supply a semicolon separated list of UDLs in place of a single UDL when connecting to the server. If the client cannot connect to the first UDL on the list, the next is tried and so on until a valid connection is made.

If the server should die during the sending or receipt of messages, the software will try to connect to a UDL on the list and continue on. Any subscriptions are propagated to the new server. Of course, a client who has a subscription will potentially miss messages sent during the brief time it is not connected. Take note that any subscribeAndGet() or sendAndGet() calls will NOT failover, only send(), syncSend(), subscribe(), and unsubscribe() will failover.

The semicolon separated list of UDLs should place the preferred UDL(s) first since it will be given higher priority. If a client is connected to a UDL which is not first in the list and that connection fails, the software will attempt to establish a connection starting with the first UDL on the list and work its way from there.

STARTING THE CMSG DOMAIN SERVER

Some of the additional details of failing over are controlled by the tags given in the UDLs used in connecting (see the table above). Each UDL may specify one of its tag/val pairs as “failover=cloud” which allows failing over to the cloud (if the server is a member of one). As long as there is a viable server in the cloud, failovers will only happen to servers in that cloud (as long as they both shall live). If, on the other hand, there are no other servers in the cloud, failovers will continue to the next UDL in the list originally given while connecting. Again, if the client was not at the head of that list at the time of failure, the first UDL is tried again, and the software makes its way down the list from there (but skipping the failed UDL). If the tag/val is specified as “failover=cloudonly”, failovers will only happen to servers in the same cloud. If no other servers exist in that cloud, the client’s connection is broken for good – no more failovers. Finally, if the user specifies “failover=any”, then the client works its way down the UDL list. If a particular UDL in that list specifies “failover=cloud”, then that’s what happens. If it specifies “failover=cloudonly”, that’s what happens from there.

Wait! There are other complications in failovers to take note of. The tag/val pair of “cloud=any” specifies that if failing over to a server in the cloud, it may choose any member of that cloud. However, the tag/val pair of “cloud=local” means that failover will first go to a server in that cloud running on the same host as the client. If there are no local servers available, failover will happen to a remote server.

Let me add a final word here on tag/val pairs which involve the performance of both clients and servers. In order to make the server perform better when serving large numbers (100’s to 1000’s) of clients, specifying “regime=low” can greatly help performance of the server (and therefore all the attached clients as well). What this tells the server is that “hey, I’m a client that makes few demands” and so the server diverts fewer resources to that client. For any client that sends and receives normal size messages at no more than 10-20 Hz, it would help to use “regime=low”.

8. Utilities and Example Programs

A number of general purpose utility applications are provided. These are fairly simple programs, and may easily be customized. Java utilities require Java 1.5 or higher.

8.1. *cMsgLogger*

The *cMsgLogger* logs messages that match subject/type to the screen, a file, and/or a database. Similar functionality exists in the File domain, and in the LogFile and Queue subdomains:

```
$ java org.jlab.coda.cMsg.apps.cMsgLogger -h
Usage:
  java cMsgLogger [-name name] [-descr description] [-udl domain]
                  [-subject subject] [-type type]
                  [-screen] [-file filename]
                  [-url url] [-table table] [-driver driver] [-account account]
                  [-pwd password]
                  [-debug] [-verbose]
```

where udl is required, subject and type default to * and *, and display to the screen is default. url is a JDBC url, table is the name of the table to use, and driver, account, and password are used when connecting to the database.

8.2. *cMsgQueue*

The *cMsgQueue* utility queues messages to either a database or file-based queuing system, same as the Queue and FileQueue subdomains. Some differences are that in the Queue and FileQueue subdomains all messages are queued, whereas here only messages that match subject/type, specified on the *cMsgLogger* command line, get queued. Also, here only the creator and user-settable fields are queued, whereas all fields are stored in the two subdomains. Finally, in the subdomains the *cMsg* server does the work, whereas here the *cMsgQueue* application does the work.

Clients retrieve from the queue by executing the *sendAndGet()* method, where the message must be sent to *getSubject/getType*, also specified on the command line. The response message returned by the *sendAndGet()* method is taken off the head of the queue.

To run the cMsgQueue application:

```
$ java org.jlab.coda.cMsg.apps.cMsgQueue -h
Usage:
  java cMsgQueue [-name name] [-descr description] [-udl domain]
                  [-subject subject] [-type type]
                  [-queue queueName]
                  [-getSubject getSubject] [-getType getType]
                  [-dir queueDir] [-base fileBase]
                  [-url url] [-driver driver] [-account account]
                  [-pwd password] [-table table]
```

where udl is required, and queueName, subject, and type default to “default”, “*” and “*”. name defaults to “cMsgQueue:queueName”, and getSubject and getType default to name and “*”.

For file queuing, queueDir specifies the directory to hold the queue files, and fileBase specifies the base for all file names (default is “cMsgQueue_queueName”).

For database queuing url and driver must be specified, and account and password may be required by the database. table defaults to “cMsgQueue_queueName”. One instance of cMsgQueue can queue to either a file or database, not both.

8.3. cMsgGateway

The cMsgGateway implements simple inter-domain communication for domains that support the send() and subscribe() messaging API functions. The cMsgGateway connects to two domains and subscribes to the same subject/type combination in each. Messages that match the subscription criteria in one domain are cross-posted to the other.

Note that a number of message fields get reset when the gateway forwards or cross-posts the message (senderTime, senderHost, etc.) Unchanged are the creator, subject, type, text, userInt, and userTime fields.

```
$ java org.jlab.coda.cMsg.apps.cMsgGateway -h
Usage:
  java cMsgGateway [-subject subject] [-type type]
                  [-name1 name1] [-udl1 udl1] [-descr1 descr1]
                  [-name2 name2] [-udl2 udl2] [-descr2 descr2]
                  [-debug]
```

where udl1 and udl2 are required, and the remainder are optional. Subject and type default to * and * (i.e. subscribe to all subjects and types), and both name1 and name2 default to cMsgGateway.

8.4. *cMsgCAGateway*

(this utility is not completed yet...ejw, 17-feb-2005)

The *cMsgCAGateway* utility serves out client data published via *cMsg* messages as Channel Access or EPICS channels, and thus forms a bridge between the *cMsg* and EPICS worlds. It is located in the *xxx* subdirectory.

The *cMsgCAGateway* is written in C++ and uses the Portable Channel Access Server library. Upon startup it reads a configuration file containing a list of channels to serve, as well as information describing what subjects to subscribe to, how to extract the channel data from the messages, etc. The gateway can also create and serve out new channels on the fly. The initial version serves out read-only data (i.e. no support for *CA put()* yet), but read-write support is planned for a future release.

To start the *cMsgCAGateway*:

```
$ cMsgCAGateway -h
Usage:
cMsgCAGateway [-name name] [-udl udl] [-descr descr]
               [-cfg config_file] [-debug]
```

Note that due to channel access limitations only one *cMsgCAGateway* can run on a node at a time.

8.5. *cMsgAlarmServer*

The *cMsgAlarmServer* logs alarm messages to a database, file, or to the screen. The server subscribes to a special subject, set on the command line, and logs specially formatted alarm messages.

The server looks at three fields in the incoming alarm message, other than the subject. The *type* field contains the alarm channel name, an arbitrary string. The *userInt* field contains the alarm severity, an arbitrary integer (e.g. you might use 0,1,2,3 for ok,warn,error,severe_error). Finally the *text* field contains an arbitrary string that is logged along with the channel name, alarm time, and severity.

The server can simultaneously log to a database, file, or the screen, but the nature of the logging is not the same for all three, as the latter two are effectively write-only. Thus for the file or screen the server simply logs the alarm information sequentially.

Database logging is more sophisticated, and four modes are possible, each using a different table. The first two modes record full or partial histories, and the tables may contain many entries per channel. In the second two modes only one entry per channel is kept. Any or all modes may be active simultaneously.

UTILITIES AND EXAMPLE PROGRAMS

In “fullHistory” mode all messages are logged. In “history” mode only severity changes are logged. I.e. if a message for a channel arrives with severity 1 but the channel is already has severity 1, then the message is ignored.

In “change” mode the latest severity state of each channel is logged, and the time is only modified if the channel changes severity state (analogous to history mode). In “latest” mode only information from the most recent alarm message is kept. To understand the difference between the two modes, imagine a channel that is monitored every minute and has been in severity 0 for a week. In change mode the time field is set to a week ago, while in latest mode it is set to one minute ago.

To start `cMsgAlarmServer`:

```
$ java cMsgAlarmServer -h
Usage:
  cMsgAlarmServer [-name name] [-udl udl] [-descr descr]
                  [-subject alarmSubject]
                  [-screen] [-file filename] [-noAppend]
                  [-fullHistory fullHistoryTable] [-history historyTable]
                  [-change changeTable] [-latest latestTable]
                  [-url url] [-driver driver] [-account account]
                  [-pwd password]
                  [-force] [-debug]
```

where `name` is the unique name of the server (default `cMsgAlarmServer`), `udl` denotes the domain to connect to, default `descr` is “`cMsg Alarm Server`”, `alarmSubject` is the subject to subscribe to (default `cMsgAlarm`), `filename` is the name of the file to log messages to (default is append mode, use `-noAppend` to force opening of a new file), `url` is the database url, `driver` is the database driver class, and `account` and `password` may be required by the database. The four table names correspond to the four logging modes described above. The tables are created if they do not exist. If a table, `filename`, or `screen` is not specified that mode is inactive.

8.6. *cMsgCommand*

```
$ cMsgCommand -h
Usage:
  cMsgCommand [-u udl] [-n name] [-d description] [-sleep sleepTime]
              [-s subject] [-type type] [-i userInt] [-text text]
```

`cMsgCommand` is a C++ command line utility that sends a message based on command line parameters. Only the subject, type, `userInt`, and text fields may be set. `sleepTime` sets how long after sending the program disconnects (units in microsec, default 1000).

8.7. *cMsgReceive*

UTILITIES AND EXAMPLE PROGRAMS

```
$ cMsgReceive -h
usage:
    cMsgReceive [-udl udl] [-n name] [-d description] [-s subject] [-t type]
```

cMsgReceive is a C++ command line utility that subscribes to a subject/type combination and prints a notice when messages arrive. It is a much simplified version of cMsgLogger.

8.8. *Example Programs*

The cMsg distribution contains a number of Java example programs in java/org/jlab/coda/cMsg/apps, and C/C++ examples in the src/examples and src/execsrc directories (relative to top level directory).

The utility programs described above, also in the Java or C/C++ directories, should be useful as examples as well.

9. Client and Server Control and Monitoring

9.1. Control

API calls exist to implement selective shutdown of clients and servers. The default client shutdown handler causes the client to exit. Programmers can override this behavior by supplying a custom handler that, e.g., causes the client to simply disconnect from the cMsg system rather than exiting. See the API docs for details.

9.2. Monitoring

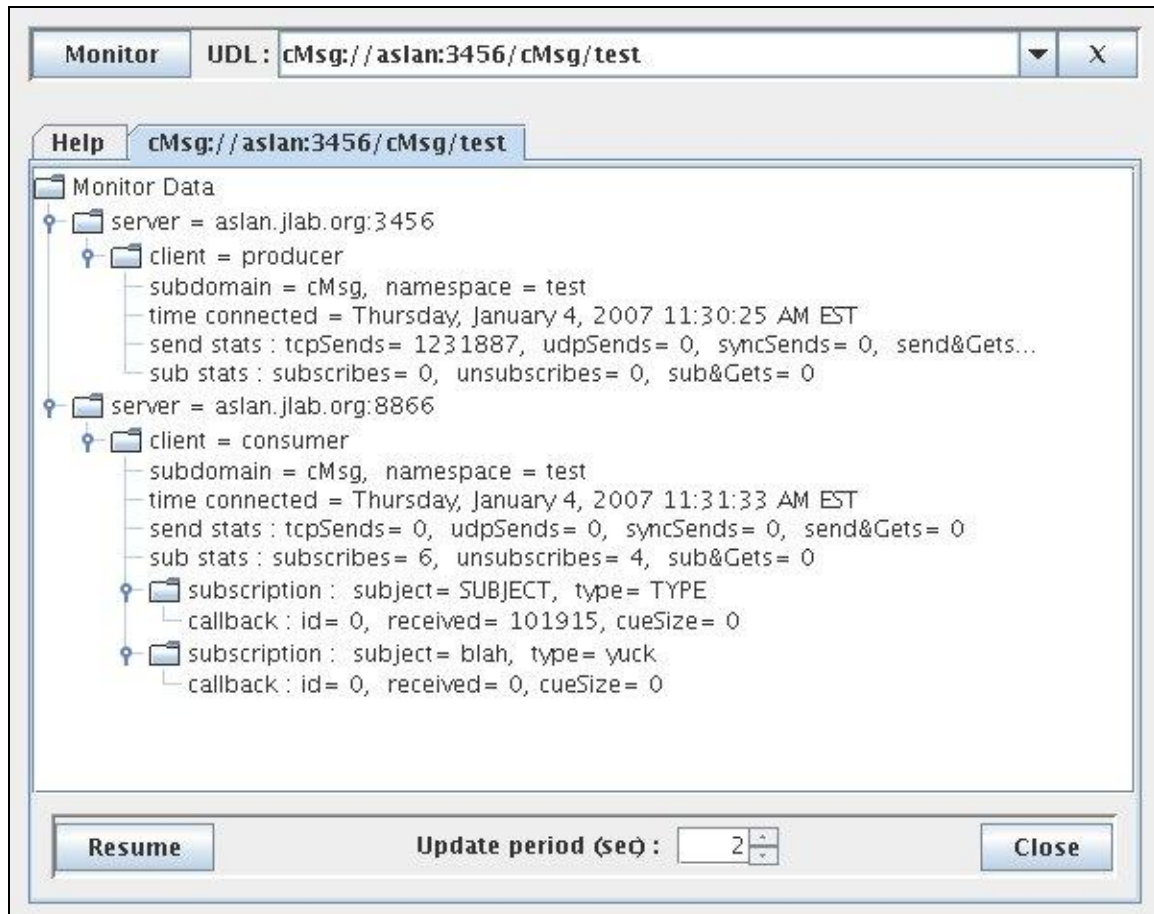
Monitoring is implemented through the **cMsgMonitor**(void *domainId, const char *command, void **replyMsg) function in C and the **monitor**(String command) method in Java, and is completely dependent on the domain implementation. See the API docs for details.

The only monitoring available to date is for the cMsg domain with the most complete information available for the cMsg subdomain. The monitoring data is collected in xml format and can be accessed either through text-based programs which print the xml or through a Java GUI which presents data in a tree.

The “monitor.c” program is an example of how to print out the xml using a C program and the Java class apps.cMsgMonitor is an example of how to do that in Java. However, to get a nice monitor GUI run the following:

```
java org/jlab/coda/cMsg/cMsgDomain/cMsgMonitor/Monitor
```

and the following GUI will appear:



It's fairly self-explanatory. Just type in the UDL, hit "Monitor", and a new tab will be added with all the monitoring information about the cMsg cloud of servers and all their clients. The monitoring can be paused/resumed, closed, and the update period can be changed.

10. Java Tutorial

An application can connect to many different domains, and publications and subscriptions in different domains are independent.

10.1. Connect to a domain

```
import org.jlab.coda.cMsg.*;
import org.jlab.coda.cMsg.cMsgException;

cMsg cMsgSys;          // the cMsg system object
try {
    cMsgSys = new cMsg(myUDL,myName,myDescr);
    cMsgSys.connect();
} catch (cMsgException e) {
    e.printStackTrace();
}
```

To disconnect from the domain:

```
cMsgSys.disconnect();
```

After calling `disconnect()`, all subscriptions and other connection information are lost. Reconnections and failovers are possible in the cMsg domain (see 5.4.1).

Note that most cMsg calls throw cMsgException, so they must be in try blocks, as shown above. Below I do not include the try blocks for clarity.

10.2. Create a message

```
// create message
cMsgMessage msg = new cMsgMessage();

// set regular fields
int myInt = 1;
long myTime = System.currentTimeMillis();
String mySubject = "s", myType = "t", myText = "txt";

msg.setSubject(mySubject);
msg.setType(myType);
```

```
msg.setUserInt(myInt);
msg.setUserTime(myTime);
msg.setText(myText);

// add a payload item (array of strings)
String[] myStrings = new String[]{"one", "two", "three"};
cMsgPayloadItem item1 = new cMsgPayloadItem("STR_ARRAY", myStrings);
msg.addPayloadItem(item1);

// add another payload item (cMsg message)
cMsgMessage myMsg = new cMsgMessage();
myMsg.setSubject("sub");
myMsg.setType("type");
cMsgPayloadItem item2 = new cMsgPayloadItem("MSG", myMsg);
msg.addPayloadItem(item2);
```

10.3. Send a message

To send a message call send then flush the outgoing message queue. Note that the system is free to flush the queue at will and in the cMsg domain, flush actually does nothing.

```
cMsgSys.send(msg);
cMsgSys.flush(null);
```

To synchronously send a message:

```
int status = cMsgSys.syncSend(msg);
// The returned status depends on domain and/or subdomain,
// and is zero for cMsg domain & subdomain.
```

where the nature of any failure is indicated in the thrown exception. The returned value is dependent on the domain or in the case of the cMsg domain, it depends on the subdomain. For the subdomain normally used, cMsg, the return value is always 0.

10.4. Subscriptions

To subscribe and unsubscribe with callback and user object:

```
import org.jlab.coda.cMsg.cMsgSubscriptionHandle;
cMsgSubscriptionHandle handle = cMsgSys.subscribe(mySubject, myType, myCB,
                                                    myUserObject);

cMsgSys.unsubscribe(handle);
```

where the callback class is:

```
import org.jlab.coda.cMsg.cMsgCallbackAdapter;
class myCB extends cMsgCallbackAdapter {
    public void callback(cMsgMessage msg, Object userObject) {
        System.out.println("Subject is:      " + msg.getSubject());
        System.out.println("Type is:         " + msg.getType());
        System.out.println("Text is:          " + msg.getText());
    }
}
```

```
}
```

and the `userObject` may be anything and exists solely for the programmer's convenience. The `handle` object is used for unsubscribing and interacting with the subscription. In the `cMsg` domain the received messages are queued and delivered serially to the callback in the order received. Configuration options (overriding of various methods) allow for parallelizing callback processing into multiple threads, discard of messages after a maximum number are waiting to be processed, etc. See the API docs for details.

To enable message receipt and delivery to callbacks:

```
cMsgSys.start();
```

To disable message receipt:

```
cMsgSys.stop();
```



ATTENTION: No messages will be delivered at all unless the `start()` method is called!

10.5. Synchronous methods

The `subscribeAndGet()` method performs a synchronous one-shot subscribe. That is, it temporarily subscribes to `mySubject/myType` (existing subscriptions and callbacks are unaffected), waits for a matching message to arrive, then returns the message. If none arrives within the timeout a timeout exception is thrown.

```
long myTimeout = 1000; /* 1 sec */
cMsgMessage m = cMsgSys.subscribeAndGet(mySubject, myType, myTimeout);
// exception thrown if no message arrived within timeout
```

The `sendAndGet()` method synchronously sends a message and gets a private response from the receiver:

```
long myTimeout = 1000; /* 1 sec */
cMsgMessage response = cMsgSys.sendAndGet(msg, myTimeout);
// exception thrown if no message arrived within timeout
```

When the receiver gets the message it has to first recognize that this is a synchronous request, then create the response message via a special method (and NOT via the usual `cMsgMessage` constructor). Receiver code might look like:

```
// ...just got a message via a callback
// ...send a response if it is a synchronous request message
If (msg.isGetRequest()) {
    cMsgMessage response = msg.response(); // create special response
    response.setSubject(mySubject);
    response.setType(myType);
    response.setText(myText);
    cMsgSys.send(response);
}
```

JAVA TUTORIAL

```
        cMsgSys.flush();  
    }
```

If two receivers synchronously respond to the message as above the first one sent gets returned by the `sendAndGet()` method in the client. The second response is treated as a normally published message.

11. C Tutorial

An application can connect to many different domains, and publications and subscriptions in different domains are independent.

11.1. *Connect to a domain*

```
#include <cMsg.h>

int stat;
void *domainId;
char *myUDL, *myName, *myDescription;

stat = cMsgConnect(myUDL, myName, myDescription, &domainId);
if (stat!=CMMSG_OK) {
    /* something is wrong */
}
```

where domainId is used in many subsequent calls to identify the connection to this particular UDL. In the following, error checking is not done for brevity and readability. To disconnect from the domain:

```
cMsgDisconnect(&domainId);
```

After calling cMsgDisconnect(), all subscriptions and other connection information are lost. Any attempt to use the domainId again will return an error. Reconnections and failovers are possible in the cMsg domain (see 5.4.1).

11.2. *Create a message*

```
/* create a message */
void *msg = cMsgCreateMessage(); /* be sure to free message when done */

/* set regular fields */
int myInt = 1;
struct timespec myTime;
char *mySubject = "s", *myType = "t", *myText = "txt";
clock_gettime(CLOCK_REALTIME, &myTime);

cMsgSetSubject(msg, mySubject);
```

C TUTORIAL

```
cMsgSetType(msg, myType);
cMsgSetUserInt(msg, myInt);
cMsgSetUserTime(msg, myTime);
cMsgSetText(msg, myText);

/* add a payload item (array of strings) */
char *myStrings[] = {"one", "two", "three"};
cMsgAddStringArray(msg, "STR_ARRAY", (const char **) myStrings, 3);

/* add another payload item (cMsg message) */
void *myMsg = cMsgCreateMessage();
cMsgSetSubject(myMsg, "sub");
cMsgSetType(myMsg, "type");
cMsgAddMessage(msg, "MSG", myMsg);
```

11.3. Send a message

To send a message call `send` then flush the outgoing message queue. Note that the system is free to flush the queue at will and in the `cMsg` domain, flush actually does nothing. Be sure to free messages when you are done with them.

```
struct timespec timeout = {0,0};
cMsgSend(domainId, msg);
cMsgFlush(domainId, &timeout);
cMsgFreeMessage(&msg);
```

To synchronously send a message:

```
int stat, statusCode;
stat = cMsgSyncSend(domainId, msg, NULL, &statusCode);
if (stat != CMSG_OK) {
    // something went wrong...
}
```

where the nature of the failure, if indicated, is domain specific, and `statusCode` holds a domain-specific return value.

11.4. Subscriptions

To subscribe and unsubscribe with callback and user arg:

```
void *myHandle;
cMsgSubscribeConfig *myConfig = cMsgSubscribeConfigCreate();
cMsgSubscribe(domainId, mySubject, myType, myCB, (void*)myUserArg, myConfig,
              &myHandle);
cMsgUnSubscribe(domainId, myHandle);
cMsgSubscribeConfigDestroy(myConfig);
```

where the callback function is:

```
void myCB(void* msg, void* userArg) {
    char *s;
    cMsgGetSubject(msg, &s); printf("Subject is:   %s\n",s);
}
```


C TUTORIAL

```
cMsgGetType(msg, &s);      printf("Type is:      %s\n",s);
cMsgGetText(msg, &s);      printf("Text is:       %s\n",s);
cMsgFreeMessage(&msg);
}
```

and the user arg, `myUserArg`, may be anything and exists solely for the programmer's convenience. The sixth argument of `cMsgSubscribe()` (`myConfig`) is a pointer to a subscription configuration created with `cMsgSubscribeConfigCreate()` and modified by the use of many other functions. Finally, the last argument, `myHandle`, is just a void pointer which is used for unsubscribing and interacting with the subscription. In the `cMsg` domain the received messages are queued and delivered serially to the callback in the order received. Configuration options allow for parallelizing callback processing into multiple threads, discard of messages after a maximum number are waiting to be processed, etc. See the API docs for details.

To enable message receipt and delivery to callbacks:

```
cMsgReceiveStart(domainId);
```

To disable message receipt use:

```
cMsgReceiveStop(domainId);
```



ATTENTION: No messages will be delivered at all unless `cMsgReceiveStart()` is called!

11.5. Synchronous Routines

The `cMsgSubscribeAndGet()` function performs a synchronous one-shot subscribe. That is it temporarily subscribes to `mySubject/myType` (existing subscriptions and callbacks are unaffected), waits for a matching message to arrive, then returns the message. Be sure to free the message when done with it.

```
void *msg;
struct timespec myTimeout = {1,0}; /* 1 sec */
stat = cMsgSubscribeAndGet(domainId, mySubject, myType, &myTimeout, &msg);
if (stat!=CMMSG_OK) {
    /* timeout or error */
}
```

The `cMsgSendAndGet()` function synchronously send a message and get a private response from the receiver:

```
void *reply;
struct timespec myTimeout = {1,0}; /* 1 sec */
stat = cMsgSendAndGet(domainId, msg, &myTimeout, &reply);
if (stat!=CMMSG_OK) {
    /* timeout or error */
}
```

C TUTORIAL

When the receiver gets the message, it has to first recognize that this is a synchronous request, then create the response message via a special function (and NOT via `cMsgCreateMessage()`). Be sure to free the message when done with it.

Receiver code might look like:

```
/*
 * Just got a message via a callback...
 * ...send a response if it is a synchronous request message
 */
if (cMsgGetGetRequest(msg)) {
    void *response = cMsgResponse(msg); /* create special response */
    cMsgSetSubject(response, mySubject);
    cMsgSetType(response, myType);
    cMsgSetText(response, myText);
    cMsgSend(domainId, response);
    cMsgFlush(domainId, &myTimeout);
    cMsgFreeMessage(&response);
}
```

If two receivers synchronously respond to the message as above the first one sent gets returned by the `sendAndGet()` method in the client. The second response is treated as a normally published message.

12. C++ Tutorial

An application can connect to many different domains, and publications and subscriptions in different domains are independent.

12.1. *Connect to a domain*

To create the cMsg system object and connect to a domain:

```
#include <cMsg.hxx>

string myUDL, myName, myDescription;
cMsg cMsgSys(myUDL, myName, myDescription)

try {
    cMsgSys.connect();
} catch (cMsgException e) {
    cout << e.toString() << endl;
}
```

To disconnect from the domain:

```
cMsgSys.disconnect();
```

After calling disconnect(), all subscriptions and other connection information are lost. Reconnections and failovers are possible in the cMsg domain (see 5.4.1). Note that most cMsg calls throw cMsgException, so they must be in try blocks, as shown above. Below I do not include the try blocks for clarity.

12.2. *Create a message*

```
// create a message
cMsgMessage msg;

// set regular fields
int myInt = 1;
struct timespec myTime;
string mySubject = "s", myType = "t", myText = "txt";
clock_gettime(CLOCK_REALTIME, &myTime);

msg.setSubject(mySubject);
```

```
msg.setType(myType);
msg.setUserInt(myInt);
msg.setUserTime(myTime);
msg.setText(myText);

// add payload item (array of strings)
string name = "STR_ARRAY", name2 = "MSG";
string strs[3] = {"one", "two", "three"};
msg.add(myName, strs, 3);

// add another payload item (cMsg message)
cMsgMessage myMsg;
myMsg.setSubject("sub");
myMsg.setType("type");
msg.add(name2, myMsg);
```

12.3. Send a message

To send a message and flush the outgoing message queue:

```
cMsgSys.send(msg);
cMsgSys.flush();
```

Many messages can be sent before flushing the outgoing queue. Note that the system is free to flush the queue at will.

To synchronously send a message:

```
int val = cMsgSys.syncSend(msg);
```

where the nature of any failure is indicated in the thrown exception. The returned value is dependent on the domain or in the case of the cMsg domain, it depends on the subdomain. For the subdomain normally used, cMsg, the return value is always 0.

12.4. Subscriptions

To subscribe and unsubscribe with callback and user object:

```
void *handle = cMsgSys.subscribe(mySubject, myType, myCB,
                                (void*)myUserObject);
cMsgSys.unsubscribe(handle);
```

where the callback class is:

```
class myCB:public cMsgCallbackAdapter {
    void callback(cMsgMessage msg, void* userObject) {
        cout << "Subject is:    " << msg.getSubject() << endl;
        cout << "Type is:      " << msg.getType() << endl;
        cout << "Text is:       " << msg.getText() << endl;
    }
}
```

and the `userObject` may be anything and exists solely for the programmer's convenience. In the `cMsg` domain the received messages are queued and delivered serially to the callback in the order received. Configuration options allow for parallelizing callback processing into multiple threads, discard of messages after a maximum number are waiting to be processed, etc. See the API docs for details.

To enable message receipt and delivery to callbacks:

```
cMsgSys.start();
```

To disable message receipt:

```
cMsgSys.stop();
```



ATTENTION: No messages will be delivered at all unless the `start()` method is called!

12.5. Synchronous methods

The `subscribeAndGet()` method performs a synchronous one-shot subscribe. That is, it temporarily subscribes to `mySubject/myType` (existing subscriptions and callbacks are unaffected), waits for a matching message to arrive, then returns the message. If none arrives within the timeout an exception is thrown.

```
struct timespec myTimeout = {1,0}; /* 1 sec */
cMsgMessage m = cMsgSys.subscribeAndGet(mySubject, myType, &myTimeout);
// timeout exception thrown if no message arrives within timeout
```

The `sendAndGet()` method synchronously sends a message and gets a private response from the receiver:

```
struct timespec myTimeout = {1,0}; /* 1 sec */
cMsgMessage response = cMsgSys.sendAndGet(msg, &myTimeout);
// timeout exception thrown if no message arrives within timeout
```

When the receiver gets the message it has to first recognize that this is a synchronous request, then create the response message via a factory (and NOT via the usual `cMsgMessage` constructor). Receiver code might look like:

```
// ...just got a message via a callback
// ...send a response if it is a synchronous request message
If (msg.isGetRequest()) {
    cMsgMessage *response = msg.response(); // create special response
    response->setSubject(mySubject);
    response->setType(myType);
    response->setText(myText);
    cMsgSys.send(response);
    cMsgSys.flush();
}
```

If two receivers synchronously respond to the message as above the first one sent gets returned by the `sendAndGet()` method in the client. The second response is treated as a normally published message.



A. Getting, Building, and Installing cMsg

You must install Java version 1.5 or higher if you plan to run the cMsg server. If you only plan to run C/C++ clients (i.e. if you will use someone else's cMsg server) you can skip the Java installation. If you only plan to use Java domains and clients you can skip the C/C++ installation.

1) Getting cMsg

The cMsg package, including documentation, can be found in the Downloads section on the JLab Data Acquisition Group CODA wiki at <http://codajlab.org>. To install cMsg, download the cMsg-3.x.tar.gz file (or whatever version happens to be current) and untar it. This will give you a full cMsg distribution with the top level directory being "cMsg-3.x". The documentation, in the "doc" subdirectory, contains User's and Developer's Guides (word, pdf), javadoc for the Java API (html), and doxygen directories containing C/C++ API docs (html). The javadoc and doxygen documentation needs to be generated by programs of the same name. Javadoc comes with any java distribution and doxygen is available at <http://www.doxygen.org>.

At this point there are 2 options for building the C/C++ code (temporarily while the CODA group evaluates each of them and settles permanently on one or the other). One build system, BMS, is based on *make*, while the other uses *scons* and *python*. Note that currently only Linux, Solaris, and Darwin (Mac OSX) operating systems are supported. The libraries and executables are placed into the \$INSTALL_DIR/<arch>/lib and bin subdirectories (eg. ...Linux-x86_64/lib). Be sure to change your LD_LIBRARY_PATH environmental variable to include the correct lib directory.

2) Compiling C/C++ code

a) Using SCons

From the SCons website: "SCons is an Open Source software construction tool -- that is, a next-generation build tool. Think of SCons as an improved, cross-platform substitute for the classic Make utility with integrated functionality similar to autoconf/automake and compiler caches such as ccache. In short, SCons is an easier, more reliable and faster

GETTING AND INSTALLING CMSG

way to build software”. SCons is written in python, thus to use this build system with cMsg, both python and SCons packages need previous installation. If your system does not have one or the other, go to the <http://www.python.org/> and <http://www.scons.org/> websites for downloading.

The SCons files are already included as part of the cMsg distribution. To compile, the user needs merely to run “scons” in the top level cMsg directory. To compile and install libraries and header files, first define either CODA_HOME or INSTALL_DIR and then (equivalent of “make install”) run “scons install”.

The following is a table of SCons compilation options:

scons option	action
-h	print out help
-c	remove all generated files
install	make C and C++ library code and general-purpose binaries; place libraries & binaries in architecture-specific directories and headers in an include directory
examples	make C and C++ executable example code; place executables in architecture-specific directory
tar	create a tar file (cMsg-3.x.tar.gz) of the cMsg top level directory and put in ./tar directory
doc	generate html documentation from javadoc and doxygen comments in the source code and put in ./doc directory
-c install	remove all installed headers, libraries, and executables for C, C++ and all generated files
--prefix=<dir>	use base directory <dir> when doing install
--dbg	compile with debug flag
--32bits	compile 32bit libraries & executables on 64bit system
--vx	cross compile for vxworks

b) Using make

The necessary makefiles for building cMsg are contained in an external package called BMS. BMS can be downloaded from the same ftp directory as cMsg. To get a complete look at this makefile system and how to use it to compile cMsg, **read the BMS documentation**. The following is a quick look at how to compile cMsg just to get you going but skips over many details.

After BMS.tar.gz has been downloaded and untarred, set the environmental variable BMS_HOME to its installation directory and the variable CODA_HOME or INSTALL_DIR to the directory which will hold the compiled cMsg libraries, executables, headers and other files. Change directories to “cMsg” and make everything by typing “make install”. Just doing a “make” at this point runs into trouble since the

GETTING AND INSTALLING CMSG

makefiles look into the install directory for the libraries when making the executables (yes, I know it's inconvenient, but if you want convenience use SCons to make cMsg). Note that currently only Linux, Solaris, and Darwin (Mac OSX) operating systems are supported for the making of all the C and C++ code.

```
# download BMS.tar.gz into myDir
# download cMsg-3.x.tar.gz into myDir
$ cd myDir
$ tar -fxz BMS.tar.gz
$ setenv BMS_HOME myDir/BMS
$ mkdir cmsg_install
$ setenv INSTALL_DIR myDir/cmsg_install
$ tar -fxz cMsg-3.x.tar.gz
$ cd cMsg-3.x
$ make install
```

This will make everything except docs – directories, java, C, and C++. There are other options for making cMsg. The following is a table of the main makefile possibilities:

make command	action
make	make all C/C++ code including creating necessary directories but does not create documentation
make help	print out help
make env	print out environmental variables
make install	make C/C++ code and place libraries and executables in architecture-specific lib and bin directories and headers in include directory
make uninstall	remove all installed headers, libraries, and executables for C/C++
make relink	delete libraries and executables, and relink object files
make tar	create a tar file (cMsg-3.x.tar.gz) of the cMsg top level directory and put in ../tar directory
make clean	clean C/C++ code in ../src subdirectories by removing libraries, executables, object and dependency files
make distClean	make clean and remove hidden architecture-specific directories
make execClean	delete all libraries and executables
make doc	generate documentation from code - javadoc and doxygen
make mkdirs	create the directories needed to store docs, include files, and architecture-specific libraries, executables, & object files
make java	make java code and put jar file in ../build/lib directory
make javaClean	remove class files from ../build/classes directory
make javaDistClean	remove all java generated files including class, doc, and jar files

Setting the variable CODA_USE64BITS to anything (value not used) will cause flags to be set that will compile the code to be 64 bits. Note that trying to compile for 64 bits on a 32 bit machine causes an error on Solaris while Linux just creates 32 bit code. Vxworks ignores CODA_USE64BITS.

GETTING AND INSTALLING CMSG

For debugging, simply define the DEBUG environmental variable to anything (its value is ignored). This automatically gets rid of the optimization flag and adds the `-g` flag to the compilers. The debug versions of binaries are given suffixes of `_d` to distinguish them from their non-debug counterparts and so that they may share directories with them. For example, a library name `libFOO.a` will have a debug version named `libFOO_d.a`, and an executable named `BAR` will have a debug version named `BAR_d`.

3) *Compiling Java*

One can obtain the `cMsg-3.x.jar` file from the general cMsg distribution as outlined above or just download the jar file by itself. In either case, put the jar file into your classpath and run your java application. The current version needs a Java 1.5 or later JVM.

If you wish to recompile the java part of cMsg, ant must be installed on your system (<http://ant.apache.org>). First extract the package files from the general cMsg tar file:

```
# download cMsg-3.x.tar.gz into myDir
$ cd myDir
$ tar -fxz cMsg-3.x.tar.gz
$ cd cMsg-3.x
```

There are 2 ways to go from here. You can use `make` which calls ant underneath:

```
$ make java
```

or you can use ant directly simply by calling “ant” in the top level directory. To get a list of option with ant, type:

```
ant help
```

ant command	action
ant, ant compile	compile all java source code
ant help	print out usage
ant env	print out value of build file variables
ant clean	delete all class files
ant cleanall	remove build and javadoc directories
ant jar	compile and create jar file
ant install	create jar file and install into 'prefix' if given on command line by <code>-Dprefix=dir</code> , else install into <code>INSTALL_DIR</code> or <code>CODA_HOME</code> if defined (in that order)
ant all	do clean, compile, then create jar file
ant javadoc	create javadoc documentation
ant prepare	create necessary directories
ant developdoc	create javadoc documentation for developer

GETTING AND INSTALLING CMSG

Included in the ./jar subdirectory are all auxiliary jar files used by the built-in domains and subdomains. Thus to use the Channel Access or the SmartSockets (sub)domains the user must include the necessary jar files in the classpath. Check the individual package web sites for more information.

4) Using C/C++

The libraries and executables are place into the \$(ARCH)/lib and bin subdirectories (i.e. Linux-x86_64/bin and Linux-x86_64/lib). Be sure to change your LD_LIBRARY_PATH environmental variable to include the correct lib directory.

5) Building Documentation

Some of the cmsg documentation is in the source code itself and must be generated and placed into its own directory. The java code is documented with, of course, javadoc and the C/C++ code is documented with doxygen comments (identical to javadoc comments). To generate the docs with the BMS system, type “make doc” from the top level directory. To generate the docs with the scons system, type “scons doc” from the top level directory. The javadoc is placed in the doc/javadoc directory, the doxygen docs for C code are placed in the doc/doxygen/C/html directory, and the doxygen docs for C++ code are placed in the doc/doxygen/CC/html directory. To view the html documentation, just point your browser to the index.html file in each of those directories.



B. Revision History

Version	Date	Comment
0.9 Beta	24-March-2005	Original beta test version
1.0	4-May-2005	First full-release version
2.0	09-Jun-2008	Added payload fields to messages
3.0	17-Sep-2009	Streamlined network communications
3.1	16-Sep-2010	Changed network communication protocol



C. Contact Information

The Jefferson Lab Data Acquisition CODA wiki is located at <http://coda.jlab.org>. The author's email addresses are listed on the second page of this manual.