Version

1.0

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

23-March-2006

Table of Contents

Section
1

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.

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 and Solaris) 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.

Section

2

2. Messaging Basics

cMsg messages are containers that hold a number of user and system settable fields. Clients typically use the subject, type, and text fields, but there are additional user settable fields (see below). The cMsg system takes care of setting many other fields, including creator, sender, senderHost, senderTime, receiver, receiverHost, receiverTime, etc. See the API docs for a complete list of system fields.

After message creation, producers must set the message subject and type fields (strings, with empty strings allowed), and can optionally set the text (string), byteArray, userInt (integer), and userTime (Unix time) fields. 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.

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). The wildcard characters "*" and "?" are allowed in subscriptions, where "*" matches any number of characters, and "?" matches a single character.

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 and then publishes to the same subject, it will receive the message it published.

Finally, messages can be sent or received using one of the synchronous mechanisms provided in the cMsg API. See the tutorial for more details.

Section

3

3. Domains and Universal Domain Locators

Fundamental to cMsg is the notion of a domain, or messaging space. 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 within the domain; 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.

Section
4

4. Default 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.

Developers can easily implement additional Java and/or C domains.

4.1. 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.

4.2. 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.

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.

4.3. RC (run control) 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=<expid>&broadcastTO=<timeout>&connectTO=<timeout>
```

where 1) port is optional with a default of 6543, 2) host is optional with a default of 255.255.255.255 (broadcast), and may be "localhost" or in dotted decimal form, 3) the experiment id or expid is optional defaulting to the environmental variable EXPID, 4) broadcastTO is the time to wait in seconds before connect returns a timeout when a rc broadcast 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.

4.4. cMsg domain

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?spar1=val1&spar2=val2…
```

where host and optional port identify a cMsg domain server, subdomain 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", the host can be "localhost", if the port is left out it will default to port 3456, and if the subdomain is missing it will default to the cMsg subdomain.

Callbacks in the cMsg domain are run in their own threads, so be careful with static data. Also, messages arriving for different callbacks are queued separately. Thus a “high priority” subject/type combination with its own callback will be promptly run even if large numbers of messages arrive for other callbacks. The queues can also be configured to parallelize message processing via multiple threads. See the API docs for details.

5. 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.

The cMsg domain server requires Java 1.5 or later:

```
$ java org.jlab.coda.cMsg.cMsgDomain.server.cMsgNameServer -h

Usage: java [-Dport=<tcp listening port>]
            [-Dudp=<udp listening port>]
            [-D<subdomainName=<className>]
            [-Dserver=<hostname:serverport>]
            [-Ddebug=<level>]
            [-Dtimeorder]
            [-Dstandalone]
            [-Dpassword=<password>]
            [-Dcloudpassword=<password>]  cMsgNameServer

       port is the TCP port this server listens on
       udp  is the UDP port this server listens on for broadcasts
       subdomainName  is the name of a subdomain and className is the
                      name of the java class used to implement the subdomain
       server         is the name of another host on which a cMsg server
                      is running whose cMsg subdomain you want to join
                      and serverport is that server's port
       debug level has acceptable values of:
               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)
       timeorder      means messages handled in order received
       standalone     means no other servers may connect or vice versa
       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
```

The default port is 3456. Be sure to specify the port number in the UDL in your client code if you do not use the default port. Multiple cMsg servers on the same node must use different TCP and UDP ports.

The debug level allows a variety of different output error and informational messages.

The timeorder option guarantees that the server passes the received messages on to the subdomain handler objects in the order that they were received. This is necessary to avoid out of order messages. As long as the subdomain handler being used does not itself change the order, the receiving order will be preserved. None of the supplied subdomain handlers change this order. Be aware that using this option means that there will be only ONE thread processing messages per client connection. In certain circumstances this may slow down the server. (On the other hand, in other circumstances it will speed up the server).

The password option is used as a security measure. 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.

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_HANDLER" 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.

6. Default 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).

6.1. 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.

6.2. 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.

6.3. 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&
          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.

6.4. 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.

6.5. 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.

6.6. SmartSockets subdomain

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

6.7. 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.

6.8. 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
```

where namespace identifies a messaging namespace. If namespace is not supplied a default namespace is used. 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. Client names must be unique in the cMsg subdomain or in the case of the cMsg subdomain names must be unique in the namespace.

Some of the peculiarities of this UDL include being able to leave off the initial "cMsg", the host can be "localhost", if the port is left out it will default to port 3456, and if the subdomain is missing it will default to the cMsg subdomain. In addition, it is possible to find servers if their host is unknown. This is done by UDP broadcasting on the local subnets and can be specified by using "broadcast" or "255.255.255.255" for the host, while any port which is given is the UDP port the server will be listening on. If a server is found by broadcasting, that server tells the client its host and TCP port so the client can do a proper connection. All this is transparent to the user who need only supply the necessary UDL.

6.8.1. Servers

Server clouds

This is the most complex of the subdomains. Let's start out by describing the servers and how they connect together. 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, 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 3456:

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

Then the second server would be started like:

```
Java -server -Dserver=hal:3456 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.

6.8.2. 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.

Clients can chose to send 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. See the API for details.

The call to flush() does nothing in the cMsg subdomain.

Why does cMsg appear three times in the UDL?

First optionally to denote the cMsg package, next to specify the cMsg domain, and finally to specify the cMsg subdomain.

7. 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.

7.1. 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.

7.2. 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.

7.3. 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.

7.4. 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.

7.5. 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.

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.

7.6. 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).

7.7. cMsgReceive

```
$ 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.

7.8. Example programs

The cMsg distribution contains a number of Java example programs in org/jlab/coda/cMsg/apps, and C/C++ examples in the src/C and src/CC directories.

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

Section

8

8. Client and Server Control and Monitoring

8.1. 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.

8.2. Monitoring

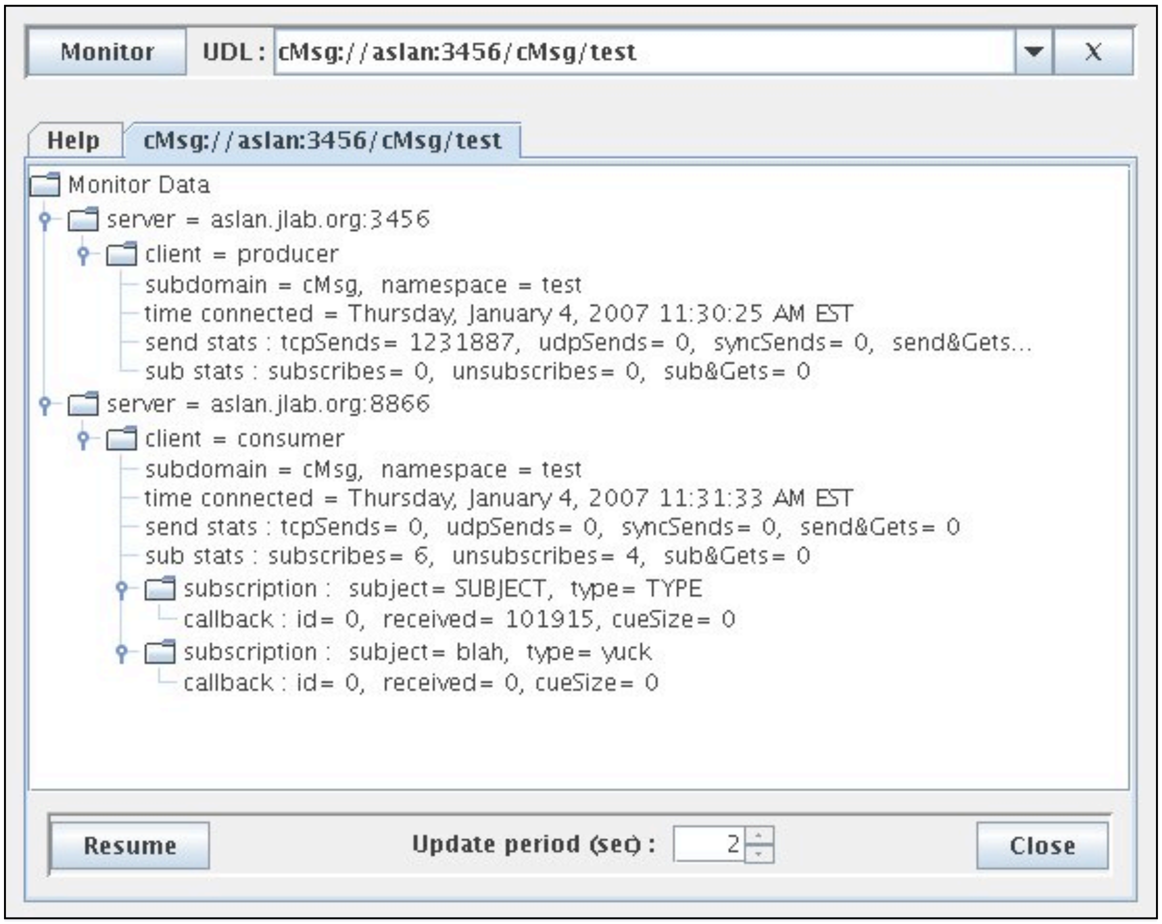
Monitoring is implemented through the cMsgMonitor(void *domainId, const char *command, void **replyMsg) function in C and the monitor(String command) method in Java, and is completely dependent on the domain implementation. See the API docs for details.

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:



UDL : cMsg://aslan:3456/cMsg/test

Update period (sec) : 2

```
cMsg://aslan:3456/cMsg/test
Monitor Data
  server = aslan.jlab.org:3456
    client = producer
      subdomain = cMsg,  namespace = test
      time connected = Thursday, January 4, 2007 11:30:25 AM EST
      send stats : tcpSends= 1231887,  udpSends= 0,  syncSends= 0,  send&Gets...
      sub stats : subscribes= 0,  unsubscribes= 0,  sub&Gets= 0
  server = aslan.jlab.org:8866
    client = consumer
      subdomain = cMsg,  namespace = test
      time connected = Thursday, January 4, 2007 11:31:33 AM EST
      send stats : tcpSends= 0,  udpSends= 0,  syncSends= 0,  send&Gets= 0
      sub stats : subscribes= 6,  unsubscribes= 4,  sub&Gets= 0
      subscription :  subject= SUBJECT,  type= TYPE
        callback : id= 0,  received= 101915, cueSize= 0
      subscription :  subject= blah,  type= yuck
        callback : id= 0,  received= 0, cueSize= 0
```

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.

Section

9

9. Java Tutorial

An application can connect to many different domains, and publications and subscriptions in different domains are independent. To create a cMsg system object and 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();
}
```

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.

To create a message and fill a few fields:

```
cMsgMessage msg = new cMsgMessage();
msg.setSubject(mySubject);
msg.setType(myType);
msg.setUserInt(myUserInt);
msg.setUserTime(new java.util.Date());
msg.setText(myText);
```

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 status = cMsgSys.syncSend(msg);
if(status != cMsgConstants.ok) {
   // something went wrong...
}
```

where the nature of the failure, if indicated, is domain specific.

To subscribe and unsubscribe with callback and user object:

```
cMsgSys.subscribe(mySubject, myType, myCB, myUserObject);
cMsgSys.unsubscribe(mySubject, myType, myCB, myUserObject);
```

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 user object 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 (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();
```

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

To disable message receipt use the stop() method.

The subscribeAndGet() method performs a synchronous one-shot subscribe:

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

where myTimeout is in integer milliseconds. subscribeAndGet() 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.

To synchronously send a message and get a private response from the receiver:

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

where myTimeout is in integer milliseconds. 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);
   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.

To disconnect from the domain:

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

After disconnect() all subscriptions and other connection information are lost.

Section
10

10. C Tutorial

An application can connect to many different domains, and publications and subscriptions in different domains are independent. To connect to the cMsg system:

```
#include <cMsg.h>

int stat;
void *domainId;
stat = cMsgConnect(myUDL,myName,myDescr,&domainId);
if(stat!=CMSG_OK) {
   /* something is wrong */
}
```

where domainId is used in many subsequent calls to identify the connection to this particular UDL.

To create a message and fill a few fields:

```
void *msg = cMsgCreateMessage();   /* be sure to free message when done */
cMsgSetSubject(msg,mySubject);
cMsgSetType(msg,myType);
cMsgSetUserInt(msg,myUserInt);
cMsgSetUserTime(msg,myTimespec);
cMsgSetText(msg,myText);
```

To send a message and flush the outgoing message queue:

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

Many messages can be sent before flushing the outgoing queue. Note that the system is free to flush the queue at will. Be sure to free messages when you are done with them.

To synchronously send a message:

```
int 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 code.

To subscribe and unsubscribe with callback and user arg:

```
   void *handle;
   cMsgSubscribe(domainId, mySubject, myType, myCB, (void*)myUserArg, config,
                 &handle);
   cMsgUnSubscribe(domainId, handle);
```

where the callback function is:

```
   void myCB(void* msg, void* userArg) {
           char *s;
           cMsgGetSubject(msg,&s);  printf("Subject is:    %s\n",s);
           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() (config) is a pointer to a subscription configuration created with cMsgSubscribeConfigCreate() and modified by the use of many other functions. Finally, the last argument, handle, is just a void pointer which is solely used for unsubscribing.

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);
```

VERY IMPORTANT: No messages will be delivered at all unless cMsgReceiveStart() is called!

To disable message receipt use cMsgReceiveStop(domainId)..

The cMsgSubscribeAndGet() function performs a synchronous one-shot subscribe:

```
   void *msg;
   stat = cMsgSubscribeAndGet(domainId,mySubject,myType,&myTimeout,&msg);
   if(stat!=CMSG_OK) {
      /* timeout or other problem */
   }
```

where myTimeout is a timespec structure. cMsgSubscribeAndGet() 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.

To synchronously send a message and get a private response from the receiver:

```
void *reply;
stat = cMsgSendAndGet(domainId,msg,&myTimeout,&reply);
if(stat!=CMSG_OK) {
   /* timeout or other error */
}
```

where myTimeout is a timespec. 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.

To disconnect from the domain:

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

After cMsgDisconnect() all subscriptions and other connection information are lost. Any attempt to use the domainId again will return an error.

11. C++ Tutorial

An application can connect to many different domains, and publications and subscriptions in different domains are independent. To create the cMsg system object and connect to a domain:

```
#include <cMsg.hxx>

cMsg cMsgSys(myUDL,myName,myDescr);       // the cMsg system object, where
try {                                     //  all args are of type string
    cMsgSys.connect();
} catch (cMsgException e) {
    cout << e.toString() << endl;
}
```

where C++ strings are use throughout (no char*). Note that most cMsg calls throw cMsgException, so they may be in try blocks, as shown above. Below I do not include the try blocks for clarity.

To create a message and fill a few fields:

```
cMsgMessage msg;
msg.setSubject(mySubject);
msg.setType(myType);
msg.setUserInt(myUserInt);
msg.setUserTime(myTimespec);
msg.setText(myText);
```

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 status = cMsgSys.syncSend(msg);
if(status != CMSG_OK) {
   // something went wrong...
}
```

where the nature of the failure, if indicated, is domain specific.

To subscribe and unsubscribe with callback and user object:

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

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 user object 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();
```

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

To disable message receipt use the stop() method.

The subscribeAndGet() method performs a synchronous one-shot subscribe:

```
cMsgMessage m = cMsgSys.subscribeAndGet(mySubject,myType,myTimeout);
  // timeout exception thrown if no message arrives within timeout
```

where myTimeout is a timespec. subscribeAndGet() 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.

To synchronously send a message and get a private response from the receiver:

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

where myTimeout is a timespec. 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.

To disconnect from the domain:

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

After disconnect() all subscriptions and other connection information are lost.

Appendix

A

A. Getting 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 portal at http://coda.jlab.org. The documentation contains User's and Developer's Guides (pdf), javadoc for the Java API (html), and Doxygen directories containing C, C++ API docs (html, pdf, latex).

2) Installing cMsg

To install cMsg, download the cMsg-1.0.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". Change directories to "cMsg" and make everything by typing "make". 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 cMsg-1.0.tar.gz into myDir
$ cd myDir
$ tar -fxz cMsg-1.0.tar.gz
$ cd cMsg
$ make
```

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

make	makes all code including creating necessary directories but does NOT create documentation

make cCode	makes C and C++ code and places libraries and executables in architecture-specific lib and bin directories
make cClean	cleans C and C++ code in cMsg/src/C, C++, regexp directories and subdirectories by removing libs, .o files, and executables
make java	makes java code
make jClean	cleans java code by removing class files
make tarfile	makes a new cMsg-1.0.tar.gz tar file and puts in top level dir
make jarfile	makes a new cMsg.jar file of all the java code and places it in cMsg/jar
make jarClean	removes cMsg.jar from cMsg/jar directory
make tarClean	removes cMsg-1.0.tar.gz from top level directory
make clean	does cClean, jClean, tarClean, and jarClean
make bClean	removes libraries and executables in lib and bin directories for the specific architecture currently being used
make distClean	does clean and bClean – essentially cleans up everything
make javadoc	creates javadoc documentation for all java code and stores in cMsg/doc/javadoc directory
make doxygen	creates javadoc-like documentation for all C and C++ code and stores in cMsg/doc/Doxygen/C/html and CC/html directories
make doc	does javadoc and doxygen – all documentation generated from code
make mkdirs	makes the directories needed to store docs, include files, and architecture-specific libs, executables, & object files

3) Using Java

One can obtain the cMsg.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, first extract the package files from the general cMsg tar file, then run make:

```
# download cMsg-1.0.tar.gz into myDir
$ cd myDir
$ tar -fxz cMsg-1.0.tar.gz
$ cd cMsg
$ make java
$ make jarfile
```

Included in the jar subdirectory are all auxiliary jar files used by the built-in domains and subdomains. Code is not provided. 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++

To create the C/C++ libraries and utilities, first expand the tar.gz archive, and then run make. After “make” is run once, all the necessary subdirectories of the distribution will be created. At that point a “make cCode” can be done to do the compilation and installation.

```
# download cMsg-1.0.tar.gz into myDir
$ cd myDir
$ tar -fxz cMsg-1.0.tar.gz
$ cd cMsg
$ make cCode
```

The libraries and executables are place into the arch/$(ARCH)/lib and bin subdirectories (i.e. arch/Linux/i686/bin and arch/Linux/i686/lib). Be sure to change your LD_LIBRARY_PATH environmental variable to include the correct lib directory.

Appendix

B

B. Revision History

Version	Date	**Comment**
0.9 Beta	2-May-2005	Original beta test version
1.0	24-March-2005	First full-release version

Appendix

C

C. Contact Information

The Jefferson Lab Data Acquisition Portal is located at http://coda.jlab.org. The author's email addresses are listed on the second page of this manual. Bug/feature requests can be made by following the "Request Tracker" link on the portal (we use the Mantis Request Tracking package). Further contact information can be found by following the YYYY link on the portal.