Version

3.5

JEFFERSON LAB

Data Acquisition Group

cMsg Developer's Guide

JEFFERSON LAB DATA ACQUISITION GROUP

cMsg Developer's Guide

Elliott Wolin
wolin@jlab.org

Carl Timmer
timmer@jlab.org

11-Nov-2013

12000 Jefferson Ave
Newport News, VA 23606
Phone 757.269.5130 • Fax 757.269.6248

Table of Contents

Section 1

1. Introduction

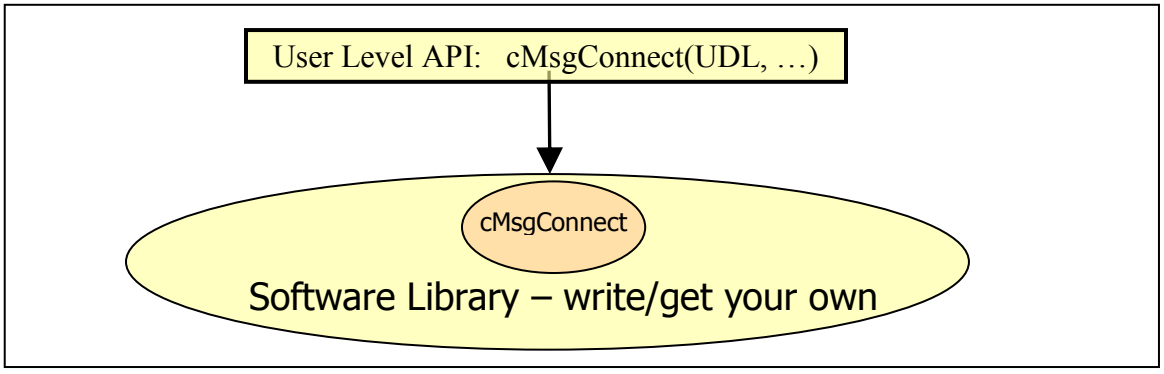
In the following we describe how to customize the cMsg package by adding new domains and subdomains. Users of this guide should be familiar with the basic concepts and strategies of the cMsg package. See the *cMsg User's Guide* for more information. As noted in the user's guide, the cMsg package can be used at a number of levels:

- 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

This guide describes how to work with the first three levels above. The user's guide discusses the fourth level, i.e. how to use the built-in messaging systems provided with the cMsg package.

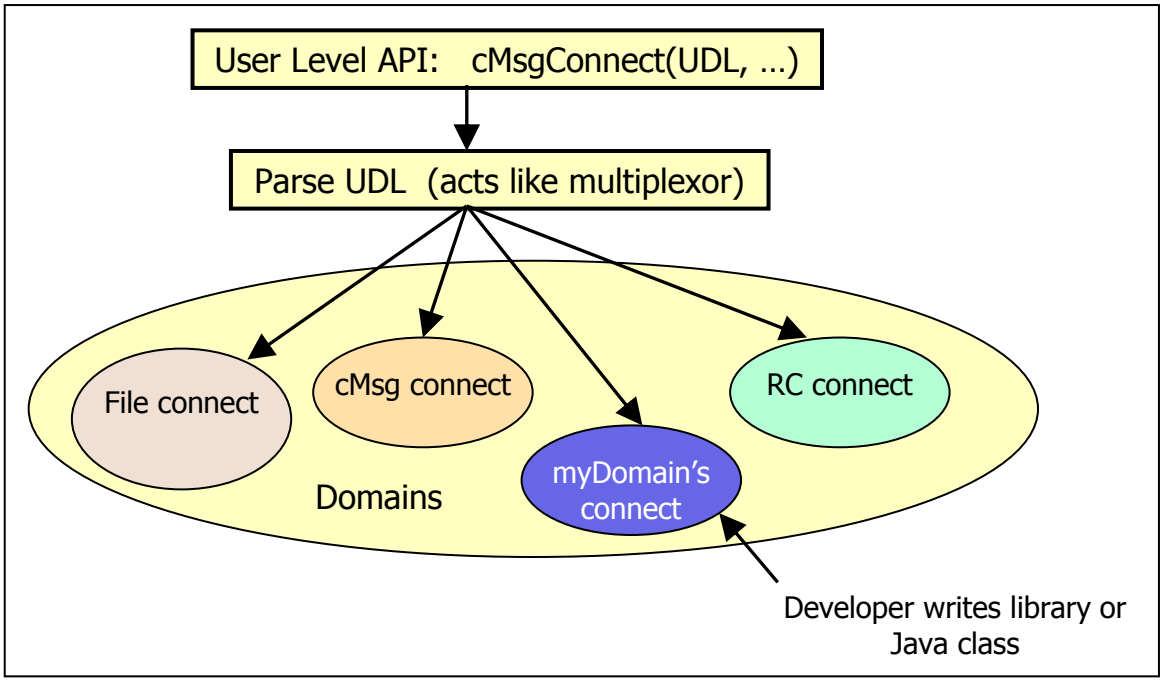
1.1. Abstract API

At the most basic level the cMsg API can simply be implemented as a façade on top of an arbitrary, asynchronous publish/subscribe or synchronous peer-to-peer messaging system. This approach may be useful since user code need not change if the underlying messaging system is changed; however, this will not be discussed further as there is little to say. Look at the diagram below to see a graphical view of how the API is implemented in this fashion. Far more powerful customizations of the cMsg package are described in the next two sections. See the API docs for details of the cMsg API.



1.2. Framework

A more interesting use of the cMsg package is as a framework for implementation of user-written domains. The cMsg client framework allows new domains to be easily added, dynamically in Java (1.5 or later), and dynamically or statically in C. The framework supports connection of clients to multiple domains, keeps track of separate callbacks and subscriptions, etc. Developers simply have to implement a basic set of messaging functions via an adapter class or interface in Java, or via a set of function pointers in C. See the graphical example below and Section 2 for details.



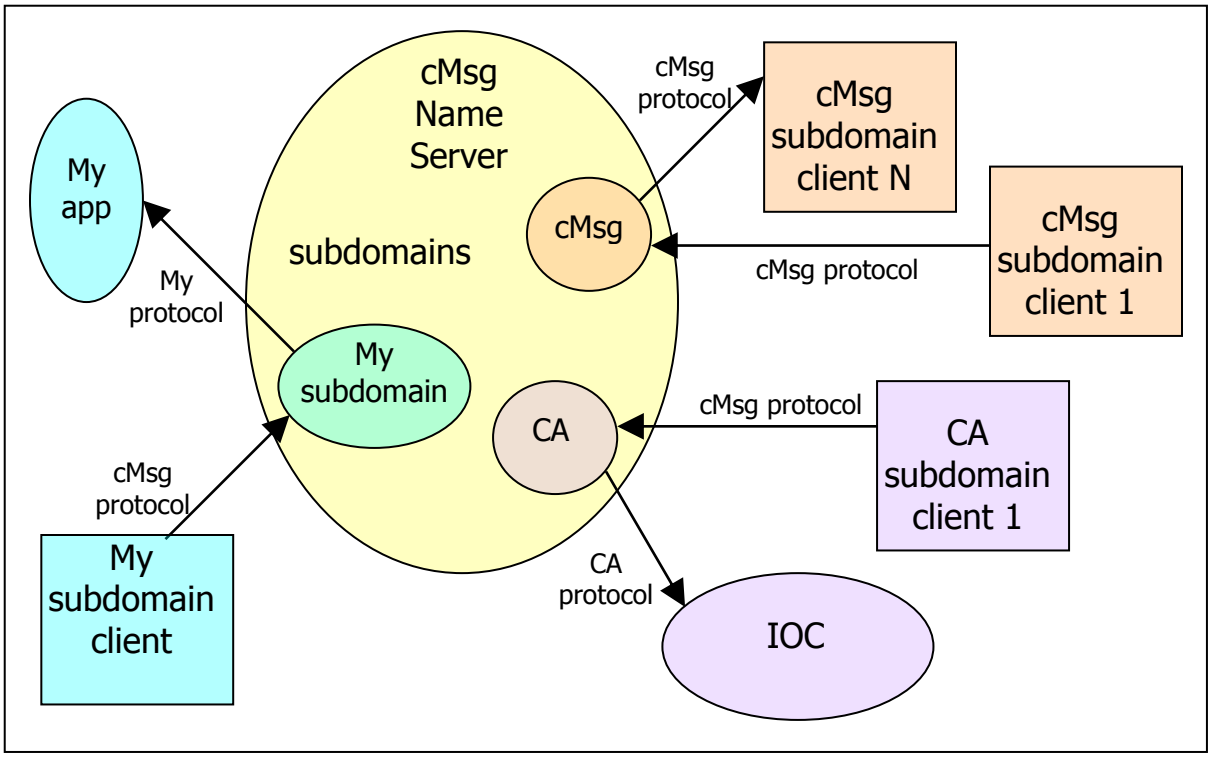
1.3. Proxy system

The most powerful customization of the cMsg package involves adding new subdomains to the built-in cMsg domain. In the cMsg domain a proxy server (provided) actually implements the messaging system and does all the work. The cMsg domain client code simply transports messages, subscription requests, etc. to the server. This is done in a completely transparent manner so that the client does not know that a remote server is involved.

Developers implement the messaging system using a subdomain interface (Java 1.5 or later) which is used by the cMsg domain server to actually do the work. Furthermore, subdomain classes can be loaded dynamically by the server as long as the class files are in the server classpath.

The major advantage of using proxy implementations is that the client nodes do not need access to licenses, libraries, etc. that are specific to the underlying messaging system.

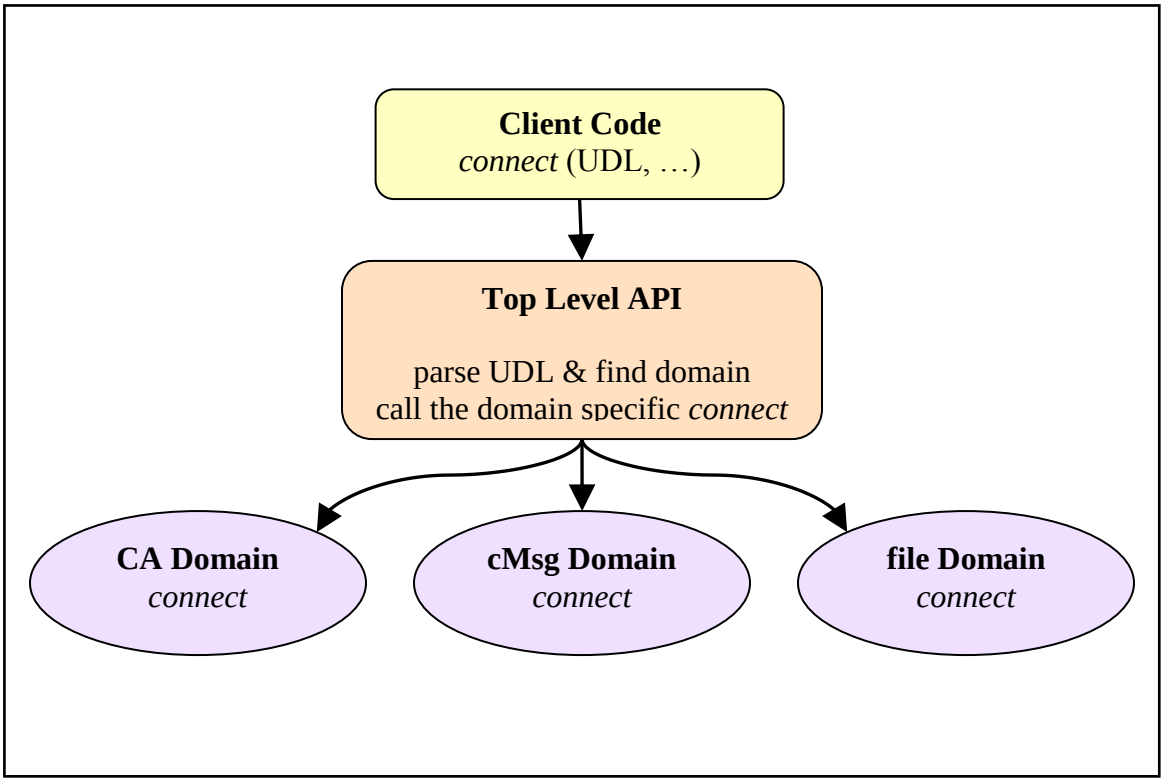
These need only be available on the machine where the server runs. For example, access to messaging systems that are not available under VxWorks or on Solaris can be made available to VxWorks or Solaris clients through the proxy server. See the example below and Section 3 for details.



Section 2

2. Framework

Let's begin by describing the general way in which both Java and C domains are structured. The cMsg package acts as a multiplexer. The top level user interface simply redirects the API calls to a specific implementation or domain. This is done simply by parsing the UDL in the "connect" call, determining the domain from the UDL and then passing the calls on to the domain level. The figure below shows graphically what happens to the connect call, but the same applies to all API calls.



There are some domains which exist only in Java and those that exist in C, C++, and Java. A domain with versions written in multiple languages must all use the same communication protocol. Another important note for the domain developer is that because clients can be multithreaded, there is the possibility that each function or method may be called simultaneously by different threads. In other words, **code which implements a domain must be thread-safe**.

2.1. Java

In order to create a new cMsg domain that plugs into the cMsg client framework, developers need to create a class that implements the cMsgDomainInterface interface or (much easier) extends the domain adapter class cMsgDomainAdapter. Both of these java files are in the <cMsg dir>/java/org/jlab/coda/cMsg/common directory. The best way to learn how to do this is to view the programmer's guide (the javadocs). The comments in the javadocs give a nice explanation about what each class and method is about. For examples of domains look at the domains listed in the table below.

Start by choosing a **unique** (case insensitive) name for your domain. Already existing names and their associated implementations are in the following table.

Domain Name	Class File Name
cMsg	**org/jlab/coda/cMsg/cMsgDomain/client/cMsg.java**
rc	**org/jlab/coda/cMsg/RCDomain/RunControl.java**
rcm	**org/jlab/coda/cMsg/RCMulticastDomain/RCMulticast.java**
rcs	**org/jlab/coda/cMsg/RCServerDomain/RCServer.java**
TCPS	**org/jlab/coda/cMsg/TCPSDomain/TCPS.java**
file	**org/jlab/coda/cMsg/FileDomain/File.java**
CA	**org/jlab/coda/cMsg/CADomain/CA.java**

Once you have a unique name, the UDL to access your domain's server will look like:

```
cMsg:<domain name>://<domain specific stuff>
```

In your domain-implementing class, you will have access to strings containing both the full UDL and just the <domain specific stuff>. Parse the UDL and pull out all necessary information. Usually information on how to reach the server is contained in the <domain specific stuff> as well as anything else you find necessary.

After the new domain is created, the question remains as to how a cMsg client can access it. The means by which this is achieved can be seen by looking inside the org/jlab/coda/cMsg/cMsg.java file. The domains listed above are provided with the software package and their location is hard coded into that file in the "createDomainConnection" method. The user is free, of course, to edit cMsg.java to include new domains and then recompile. However, an easier alternative is to give a flag to the JVM when running the client. This flag must be of the form:

```
-D<myDomain>=<myDomain'sClassName>,
```

where the specified class must be in your classpath. For example if I have a domain called "pow" and that is implemented in a java file whose full name (including package) is bam.java and my client code is in a class called crash.java, then I would run my client with a command like:

```
java -Dpow=bam crash
```

Note that domain names are not case sensitive, but dots are not allowed in domain names.

2.2. C

Similarly, developers need to provide a library that contains an appropriate set of functions that implement the API in the C language. In this case, new domains may be added either dynamically or statically.

2.2.1. Static

If adding statically, a bit of code editing must be done. In the C code directory there is a file called cMsgPrivate.h. This file contains 2 structures that must be filled – domainFunctions and domainTypeInfo. To get a good idea of how to do this, see the doxygen html pages. A good example of a functioning domain can be found in cMsgDomain.c which is the C implementation of the cMsg domain. Alternatively, a dummy domain implementation exists in src/libsrc/dummyDomain.c and can be used as a template – just fill in the functions with your code and change the "dummy" string with your domain name. This works in conjunction with a dummy client found in src/examples/dummy.c

Just follow these steps:

1) Create and fill a structure of type "domainFunctions" with pointers to all the functions implementing the cMsg API.
2) In the global space, create and fill a structure of type "domainTypeInfo" in which the first element is a string containing the name of the domain and the second element is the structure with all the function pointers. Make sure the name of this structure is globally unique.
3) Now add this structure to the cMsg.c file declaring it extern. For example,
   a. ∈xtern domainTypeInfo myDomainTypeInfo;
4) Again in cMsg.c, add it to an existing array of domainTypeInfo structures called dTypeInfo. To do this, go to the function registerPermanentDomains and add 2 lines:
   a. dTypeInfo[n].type = (char *)strdup(myDomainTypeInfo.type);
   b. dTypeInfo[n].functions = myDomainTypeInfo.functions;
5) where n is the next available integer.
6) Finally, recompile.

2.2.2. Dynamic

If adding dynamically, a shared library must be created. In the shared library, certain functions must be present. The following is a list of the necessary functions:

```
/* Prototypes of the 25 functions which implement all the cMsg tasks. */
int   cmsg_<domain>_connect(const char *myUDL, const char *myName,
             const char *myDescription, const char *UDLremainder, void **domainId);
int   cmsg_<domain>_reconnect(void *domainId);
int   cmsg_<domain>_send(void *domainId, void *msg);
int   cmsg_<domain>_syncSend(void *domainId, void *msg, int *response);
int   cmsg_<domain>_flush(void *domainId);
int   cmsg_<domain>_subscribe(void *domainId, const char *subject, const char *type,
                              cMsgCallbackFunc *callback, void *userArg,
                              cMsgSubscribeConfig *config, void **handle);
int   cmsg_<domain>_unsubscribe(void *domainId, void *handle);
int   cmsg_<domain>_subscriptionPause (void *domainId, void *handle);
int   cmsg_<domain>_subscriptionResume(void *domainId, void *handle);
int   cmsg_<domain>_subscriptionQueueClear(void *domainId, void *handle);
int   cmsg_<domain>_subscriptionQueueCount(void *domainId, void *handle, int *count);
int   cmsg_<domain>_subscriptionQueueIsFull(void *domainId, void *handle, int *full);
int   cmsg_<domain>_subscriptionMessagesTotal(void *domainId, void *handle,
                                              int *total);
int   cmsg_<domain>_subscribeAndGet(void *domainId, const char *subject,
                                    const char *type, const struct timespec *timeout,
                                    void **replyMsg);
int   cmsg_<domain>_sendAndGet(void *domainId, void *sendMsg,
                               const struct timespec *timeout, void **replyMsg);
int   cmsg_<domain>_monitor(void *domainId, const char *command, void **replyMsg);
int   cmsg_<domain>_start(void *domainId);
int   cmsg_<domain>_stop(void *domainId);
int   cmsg_<domain>_disconnect(void *domainId);
int   cmsg_<domain>_shutdownClients(void *domainId, const char *client, int flag);
int   cmsg_<domain>_shutdownServers(void *domainId, const char *server, int flag);
int   cmsg_<domain>_setShutdownHandler(void *domainId, cMsgShutdownHandler *handler,
                                       void *userArg);
int   cmsg_<domain>_isConnected(void *domainId, int *connected);
int   cmsg_<domain>_setUDL(void *domainId, const char *udl, const char *remainder);
int   cmsg_<domain>_getCurrentUDL(void *domainId, char **udl);
```

where "<domain>" is the domain name in all lowercase letters. Files containing these functions must be compiled into a shared library with the specific name libcmsg<domain>.so where domain is the domain name in all lowercase letters. In this way, a cMsg application will parse a given UDL, find the domain, and know which library to dynamically load. If the library is in the LD_LIBRARY_PATH environmental variable, then everything should be fine.

To reiterate, just follow these steps:

1) Create a file or files of the above listed functions.
2) Compile everything into a shared library with the name libcmsg<domain>.so, domain being all lowercase.
3) When compiling your cMsg application that uses this domain, link it with this shared library as well as against libcmsg.so and libcmsgRegex.so.

4) Run your application.

An example is provided in 2 files: dummyDomain.c and dummy.c. As the name indicates, the dummyDomain.c file implements an example domain which does nothing except print something out each time a function is called. The file dummy.c implements an application which uses the dummy domain. Using these 2 files as a template should make domain development quite simple.

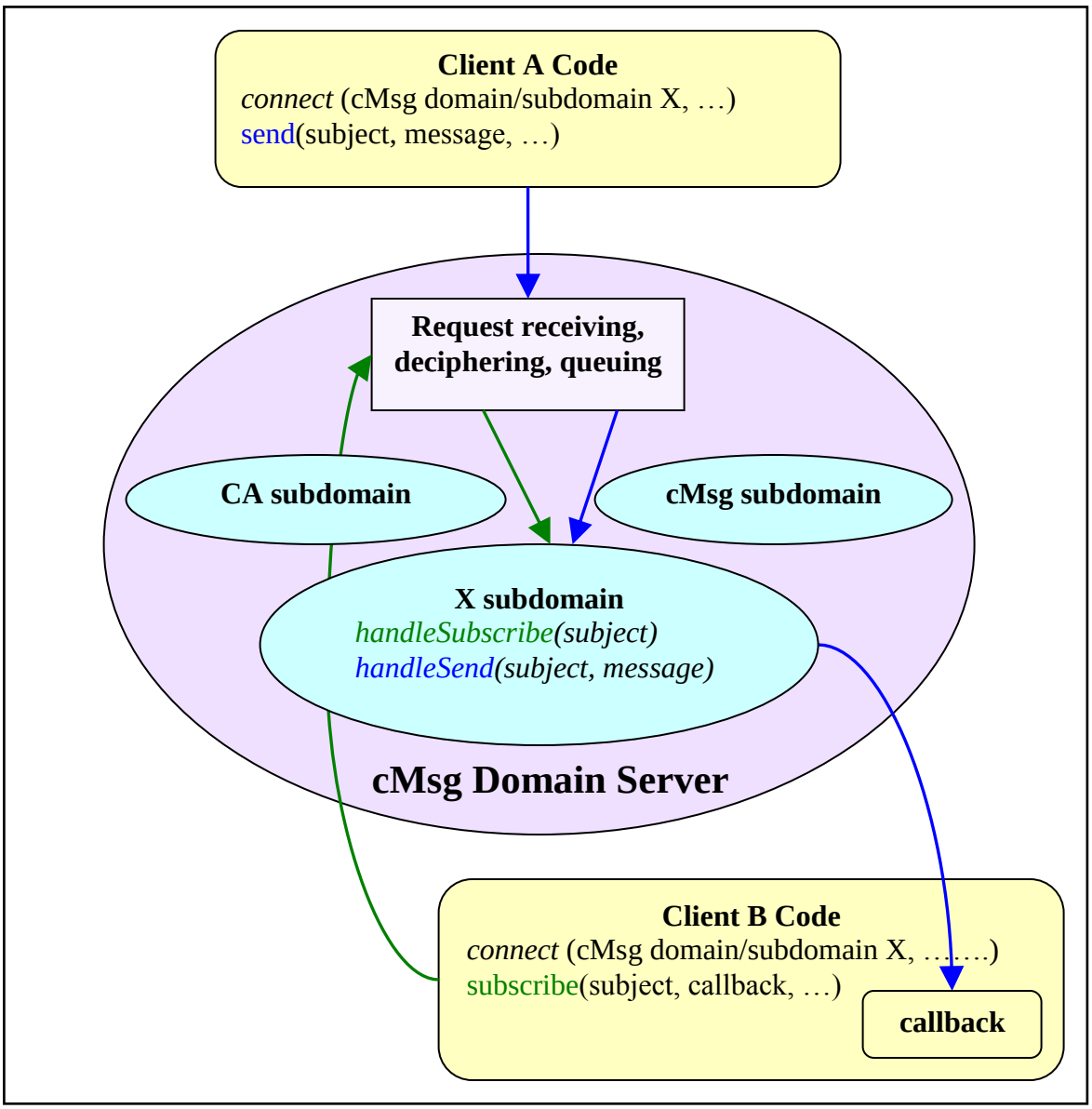
Section 3

3. Proxy Server

The cMsg domain is implemented with a server. All client API calls are directed to this server which carries out the desired actions. Because of the way this server is structured, it is also possible to use it as a proxy server. The part of the server which receives, deciphers, and queues up requests is distinct from the part which carries out the requests. In fact, for a single client, all requests are carried out by a single object which implements a specific interface – the cMsgSubdomainInterface. In actual practice it is easier to subclass cMsgSubdomainAdapter which already implements some minimum functionality in which default methods return "Not Implemented" for all messaging functions. Instead of listing and describing all methods of the interface, it's easier to read the javadoc where such is already done for a complete description of the interface.

The object that handles this real functionality we call a subdomain handler with the idea that different classes that implement this interface can handle the messages in their own way (their own subdomain). Creating a new cMsg subdomain is the most powerful way of customizing cMsg. Messages published to the cMsg domain are transparently transferred from the client to the server, then passed on to an instance of a subdomain class for transport and delivery.

The diagram below is a graphical representation of what goes on. There are 2 clients in yellow – client A and client B. The cMsg domain server is the large oval in purple which contains 1 section for receiving requests and 3 different subdomains. Each subdomain is created when a client connects to the cMsg domain server and that specific subdomain. Thus both clients have already connected to cMsg domain & subdomain X. Client B has subscribed to messages of a certain subject which is shown by green arrows and is ultimately taken care of by the handleSubscribe method of the subdomain. Client A has sent messages with that same subject and is shown with blue arrows and is ultimately taken care of by the handleSend method. Finally, client B received those messages from the subdomain handler in a callback.



The developer needs to be aware of one very pertinent fact. Clients may connect to the name server in a manner in which there will be several threads processing a single client's requests on the domain server's request queue. Thus, **all methods in the subdomain handler object must be thread-safe**. It is also possible that more than one client of a particular subdomain may be simultaneously connected, thus the subdomain handler class must be careful in handling class static data – i.e. it must be thread-safe.

There are several subdomains already existing that will serve as templates for any developer. There are subdomains for talking to Channel Access (CA), a database (DB), smart sockets (SmartSockets), queuing messages in a database (Queue), queuing messages in files (FileQueue), and logging messages to a file (LogFile). There is also a dummy subdomain (Dummy) that provides a template.

Section 4

4. Networks

4.1. Run Control Communication

Run control communication is a tricky business. Part of the difficulty is getting things to work by means of the cMsg interface, which is not a natural way to implement them. There are 3 run control domains which all work together:

1. rc domain
2. rc multicast domain
3. rc server domain

The rc domain is available in both C and Java in order to implement the rc client, while the 2 server domains (rc multicast and rc server) are only available in Java

4.1.1. RC Domain

When an rc client (rc domain) is created, the UDL given to the cMsg object is of the form:

```
rc://<host>:<port>/<expid>?multicastTO=<timeout1>&connectTO=<timeout2>
```

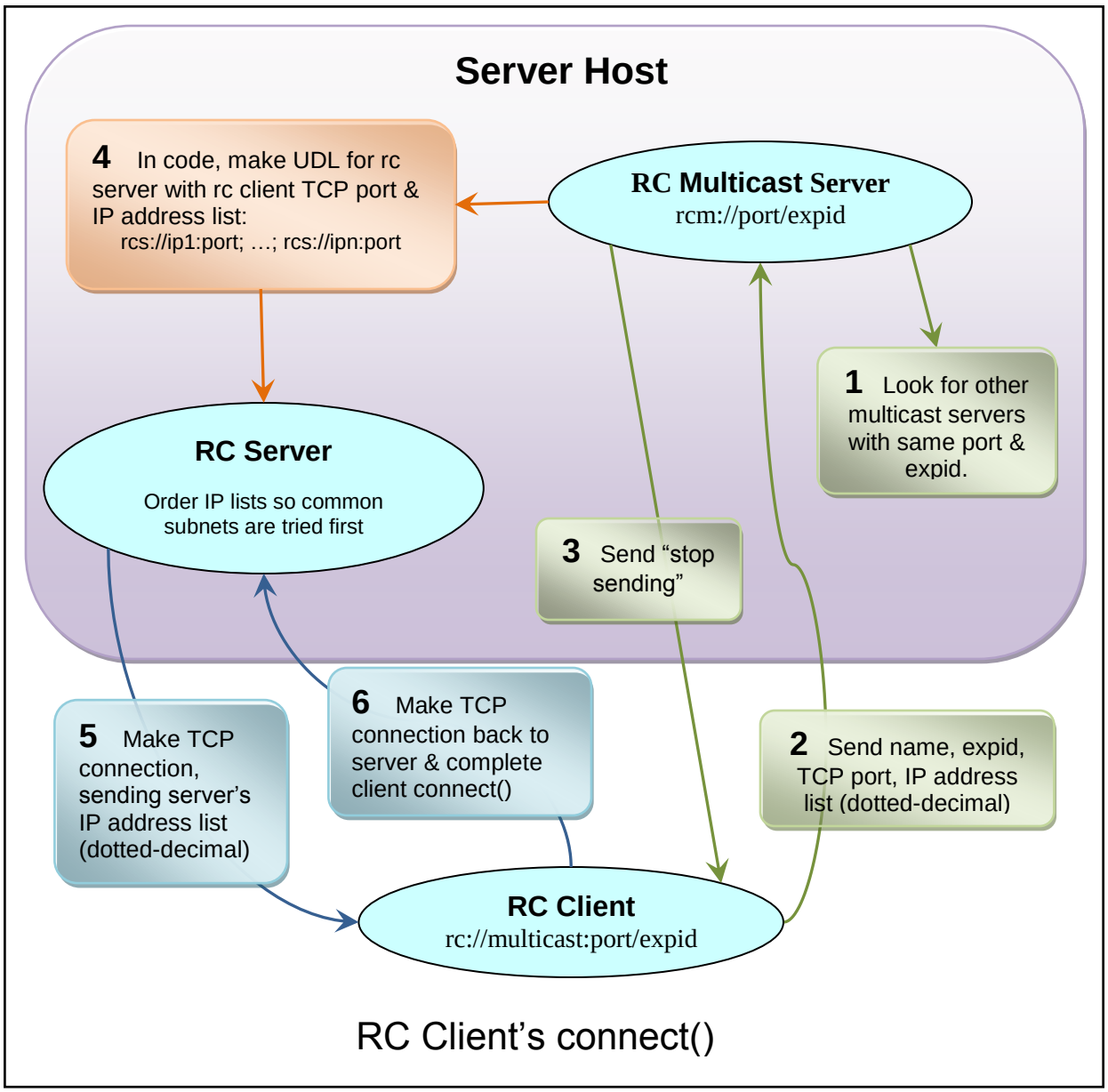
where <host> is required and may be “multicast”, “localhost”, or an ip address in dotted-decimal form. If the UDL has <host> = “multicast”, multicast packets are sent to the 239.210.0.0 address. The TCP port that the rc client is listening on is given by <port> and defaults to 45200 if not given. The value of <expid> is required and is **NOT** taken from an environmental variable. The 2 timeouts are optional. The first is the time in seconds to wait for the rc multicast server to respond and defaults to no timeout. The second is the time in seconds to wait, after the multicast server responds, until the rc server establishes a TCP connection to the client. It defaults to 5 seconds (with 0 meaning no timeout).

Client UDP packets are sent over all network interfaces that are “up”, support multicasting, and are not the loopback. These packets are staggered by ½ second to prevent the rc multicast server from being bombarded on all interfaces at once.

The rc client sends a list of its local ip addresses as strings in dotted-decimal form (address associated with the canonical name first) along with its name, expid, and TCP

port to the rc multicast server. It only sends unique, IPv4, non-loopback addresses from network interfaces which are “up”.

Below is a simplified diagram of the connection process:



4.1.2. RC Multicast Domain

Creating an rc multicast server (rcm domain) to receive the output of the rc client is done by specifying a UDL in the form:

```
rcm://<udpPort>/<expid>?multicastTO=<timeout>
```

The <udpPort> is the value of the UPD listening port which defaults to 45200 if not specified. The <expid> is required, and an optional timeout in seconds can be specified used when searching for other rcm servers with the same port and expid.

The first thing an rc multicast server does is multicast trying to locate another rc multicast server using the same port and expid. If one is found, an exception is thrown and no server started.

The rc multicast server receives a list ip addresses from the rc client and if the client's expid matches it own, places them into a cMsg message payload as a String array labeled "IpAddresses". This message also contains the rc client's TCP port and name. It is passed to all callbacks of the rc multicast server (at least one callback must be created in order to handle valid client connections).

4.1.3. RC Server Domain

These ip addresses are used in the callback or elsewhere to create an rc server object (rcs domain). The full UDL passed to the constructor of that cMsg object may be a semicolon-separated list of any number of smaller UDLs:

```
rcs://ipAddr1:port;rcs//ipAddr2:port; … rcs://ipAddrN:port
```

where "ipAddr" is a dotted-decimal format ip address of the rc client and "port" is the TCP port the rc client is listening on.

By using a list of UDLs as the full UDL itself, all rc client ip addresses can be passed to the rc server object. When parsing the full UDL, that server will extract all ip addresses and the port and find (if possible) an address on the same subnet as the server and use that as the first address in a list of rc server addresses to send to the rc client in the completion of the rc client's connect method. There should be only 1 port value used. The first port encountered in a UDL will be used as the TCP port that the rc client is listening on to receive messages from the rc server.

To complete the connect() call that the rc client makes, the rc server will send a cMsg message to it with its TCP port and a list of all its ip addresses (with the one on the same subnet first) in a payload of String array type labeled "IpAddresses". The rc client will establish a TCP connection with the rc server, trying each address in turn until a successful connection is made. Currently UDP communication is not used between rc client and server.

The way in which the rc server determines whether a client ip address is on the same subnet as one of its own ip addresses is in the following method. Each local ip address as

an array of 4, 1-byte ints and its accompanying subnet mask (a short integer) is turned into a network or subnet address. This is done by turning the 4 bytes into a 32 bit integer and anding it with the mask. The resulting 32 bit int is converted back into an array of 4, 1-byte ints which is then converted to a dotted-decimal string. This same process is done to each member of the list of client addresses. A string comparison is done and if the strings are identical, it indicates the server & client addresses are on the same subnet. The matches are placed first in a list. All non-matches are placed at the end of the list.

There are 2 lists that the rc server keeps. One is a list of server ip addresses to send to the client. The other is the list of client ip addresses. The latter is used to finish the original rc client connection call – using the common network addresses first. The former is sent to the rc client – again with the common network addresses first.

In the rc client's last part of its connect() method or routine, it creates a TCP connection to the rc server using the list of ip addresses sent to it in a cMsg message payload named "IpAddresses". This socket handles all subsequent communication between the two.