cMsg - A general purpose, publish-subscribe, interprocess communication implementation and framework

Carl Timmer, D Abbott, V Gyurjyan, G Heyes, E Jastrzembski and E Wolin

Jefferson Lab, 12000 Jefferson Ave. MS-12B3, Newport News, VA 23606, USA[1]

Email: timmer@jlab.org

Abstract. cMsg is software used to send and receive messages in the Jefferson Lab online and runcontrol systems. It was created to replace the several IPC software packages in use with a single API. cMsg is asynchronous in nature, running a callback for each message received. However, it also includes synchronous routines for convenience. On the framework level, cMsg is a thin API layer in Java, C, or C++ that can be used to wrap most message-based interprocess communication protocols. The top layer of cMsg uses this same API and multiplexes user calls to one of many such wrapped protocols (or domains) based on a URL-like string which we call a Uniform Domain Locator or UDL. One such domain is a complete implementation of a publish-subscribe messaging system using network communications and written in Java (user APIs in C and C++ too). This domain is built in a way which allows it to be used as a proxy server to other domains (protocols). Performance is excellent allowing the system not only to be used for messaging but also as a data distribution system

1. Introduction

The CODA [1] data acquisition (DAQ) package at Jefferson Lab (JLab) has been in use and under continual development for over a decade, and currently employs a number of mutually incompatible interprocess communication (IPC) systems and API's. Recently we decided to unify all interprocess communication used for control (not data) under a single API, as well as decrease the number of underlying communication packages used. This should simplify life for developers and users, and allow us to change or add new underlying IPC systems without having to modify application code.

Prior experience with publish/subscribe (pub/sub) software and its use in some experiments at Jlab, directed us to consider using it for a general API. Its simple but powerful paradigm was general enough to wrap all of CODA's existing messaging systems.

In regards to performance, we sets goals of 100's of Hz for message rate, 100's of bytes for message size, and being able to handle hundreds of clients. These requirements stem mainly from experiments at the planned JLab 12 GeV upgraded accelerator. We also avoided commercial components since we distribute software to many groups in and outside Jlab, and including commercial components can be problematic and expensive.

There are a few packages developed with similar goals. CDEV [2] is also a thin layer on top of multiple underlying IPC systems. However, the 3 main functions of the CDEV API (get, set, and

[1] Notice: Authored by Jefferson Science Associates, LLC under U.S. DOE Contract No. DE-AC05-06OR23177. The U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce this manuscript for U.S. Government purposes.

monitor) is not comprehensive enough for our needs. Abeans [3] acts as a layer on top of multiple underlying physical control systems, but the Abeans package is designed to solve a different problem than ours. Implementing a pub/sub system with DIM [4] was another possibility. Although DIM is an excellent framework, its client interface has essentially only 1 function (sendCommand) designed to send ints, doubles, and a string -- too simple to be a good match for our requirements.

Once developed, the cMsg software package met all of its initial design goals and exceeded the performance goals by a large margin. That allowed us to use cMsg not only for control applications such as runcontrol, but also as a data distribution system.

2. What exactly is Publish/Subscribe?

The asynchronous publish/subscribe interprocess communication paradigm is widely used in industry and has proven to be very powerful and successful; yet the model is extremely simple.

In asynchronous pub/sub messaging, producers create messages and then publish them to abstract subjects, in a launch-and-forget mode. Message consumers subscribe to abstract subjects and provide callbacks to handle messages as they arrive, in a subscribe-and-forget mode. Neither producer nor consumer knows of each other's existence. A single process can be both a producer and consumer. Note that multiple groups of processes can communicate without interfering with each other via simple subject naming conventions.

The asynchronous nature of this paradigm is a good match for the asynchronous nature of communication within real-time and online control systems. In this context, processes are often multi-threaded - performing multiple tasks. Control information arrives sporadically and must be handled as it arrives and on a priority basis.

3. What is cMsg?

cMsg is actually many things. It is an API, a framework, a pub/sub implementation, and a proxy system. Starting with the simplest first, we will present all of these aspects.

3.1. cMsg is an API

Any implementation of this API can be considered to be an implementation of cMsg. The following table contains a simplified, generic form of the client API.

Table 1. A table of the major client API routines (simplified) and the descriptions of what they do.

API Call	**Description**
connect(UDL, name)	Connect to cMsg system specified by UDL for client "name"
disconnect()	Disconnect from cMsg system
send(msg)	Send message asynchronously
flush(timeout)	Flush messages sent from client
syncSend(msg, timeout)	Send message and wait for server response
sendAndGet(msg, timeout)	Send message and wait for receiving client to send response
subscribe(subject, type, callback)	Subscribe to messages of given subject & type, registering callback for incoming messages
unsubscribe()	Remove subscription
subscribeAndGet(subject, type, timeout)	Subscribe to subject & type and wait for one response
start()	Start receiving messages
stop()	Stop receiving messages
monitor(command)	Synchronous call to request monitoring information

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 directed back to the original sender.

Although too many to be listed here, there are many other API routines that handle messages and allow for servers and clients to be shutdown.

3.2. cMsg is a framework

cMsg is also a framework for sending messages in multiple, underlying message systems (or **domains**) as seen in figure 1. In this capacity, the cMsg client API is implemented as a thin dispatching layer. This dispatching layer directs each API call to the underlying cMsg API of a particular domain. It is the dispatching layer that is provided by the main cMsg library (or jar file) and gets called by the user. Generally, messages published within one domain are not delivered within another domain.

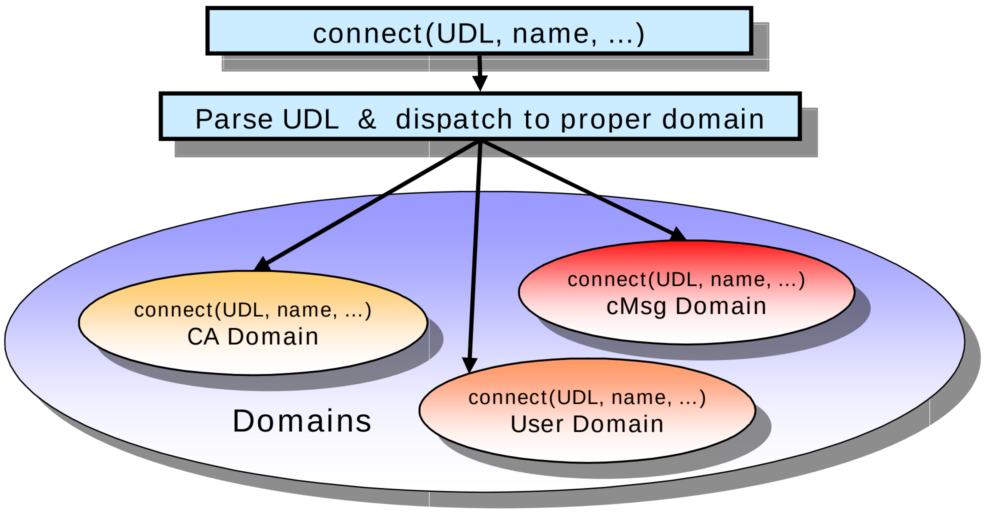

Figure 1. The cMsg client API is implemented as a thin dispatching layer to underlying messaging systems. The connect function is shown as an example.

Each domain can be identified by an http-inspired Universal Domain Locator or UDL. The general form of a UDL is:

cmsg:<domain_name>://<domain_specific_string>

where the initial "cmsg:" is optional. For example, when the user calls "connect", the dispatching layer parses the UDL to determine the domain. It then directly calls the "connect" function provided by that particular domain, passing its arguments on as is. This is done with each call into the API. In the domain's "connect", the UDL is parsed again to extract information necessary to complete the connection.

There are a number of domains that have been developed and are available for use. In C/C++ there are the cMsg, file, and rc (runcontrol) domains. In Java there are the cMsg, file, rc, and ca (channel access) domains. Aside from the cMsg domain which is described in the next section, the file domain stores messages in a file, and the ca domain allows communication with the EPICS world using its channel access communication protocol. The rc domain is only of use to implement the Jlab online runcontrol system. It is simple to create additional domains and access them for Java by putting their classes in the classpath or for C/C++ by dynamically loading a library.

3.3. cMsg is a domain / proxy server

The name “cMsg” also denotes a particular domain and is shown in figure 2. This domain uses a Java server (Java version 1.5 or later) as a proxy or broker for all interprocess communication. Clients communicate with the server using a built-in proprietary protocol, and the (possibly remote) server interacts with the underlying IPC system on the client’s behalf. Thus for example, the proxy system allows a vxWorks client to communicate with an IPC system that does not provide a vxWorks API and library. Byte-swapping and other system-dependent transformations are taken care of automatically.

To determine what to do with a client’s calls into the API, the server in the cMsg domain has behavior determined by dynamically pluggable modules which we call **subdomains**. Each subdomain is simply a Java class implementing a particular interface. The UDL given by a client is sent to the server which parses it and determines which subdomain to use.

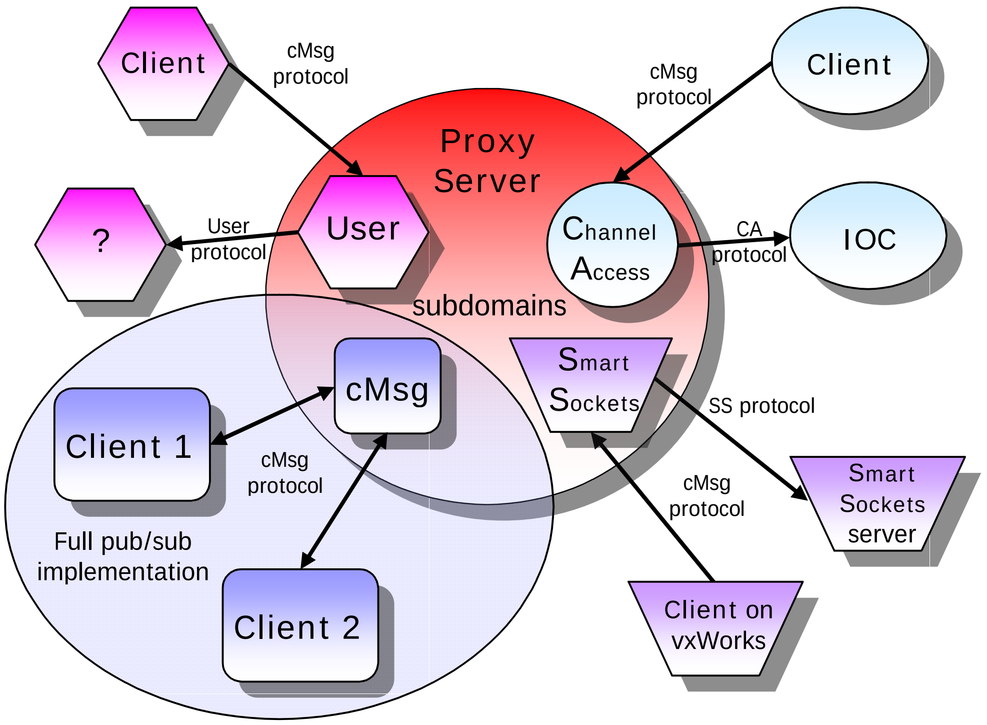

Figure 2. The client and server sides of the cMsg domain. The proxy server is written in Java. Clients can be Java, C, or C++.

The cMsg domain UDL is of the following form:

cMsg:cMsg://<host>:<port>/<subdomain_name>/<subdomain_specific_string>?tag=val&tag2=val2

which specifies the cMsg domain, and the host and port on which the cMsg domain server is running. It also specifies the subdomain to use and any additional information to be given to the subdomain. The tag-value pairs carry information that can be used to direct the behavior of the server such as specifying a password. Here we see the power of the UDL to control messaging behavior at runtime. Since a fair amount of information can be placed in a UDL, there is tremendous flexibility built in to the cMsg software package.

Often, subdomains can do exactly what domains do. There are many subdomains available for client use including: cMsg (see next section), CA (EPIC’s channel access), Database (execute database commands), SmartSockets (talk to Tipco’s pub/sub system), Queue (queues messages using database), FileQueue (queues messages using files), and LogFile (log messages in files). Many of these subdomains do not implement full pub/sub behavior and in some cases just a very small subset. Also note that each cMsg domain server can handle multiple clients simultaneously, each of which may be using a different subdomain.

3.4. cMsg is a complete publish/subscribe implementation

Again, we have used the name “cMsg”, this time to label the full featured, asynchronous publish/subscribe system implemented as a subdomain in the cMsg domain, with a few synchronous peer-to-peer mechanisms added for convenience (see figure 2). Although commercial pub/sub packages exist that could meet our needs, as mentioned earlier we wanted to avoid commercial packages if possible. Furthermore, none of the public domain packages we knew about included all the features we needed. Thus we decided to attempt implementing our own package in Java. We were surprised how quickly we were able to implement the base functionality, so we decided to continue to develop the full system in Java with client interfaces in C and C++ as well.

The Java servers in the cMsg domain have special features when using the cMsg subdomain. When started up, servers can connect to each other in what we call a server cloud so that clients connected to one server are able to send messages to and receive messages from clients connected to another server. This facilitates load-balancing. We also implemented an auto-failover feature where clients will automatically be connected to another server if their current server dies. The user simply supplies a semicolon-separated list of UDLs in its connect function. If its current connection fails it tries connecting to the other UDLs.

Monitoring is built into the cMsg subdomain. By calling the “monitor” function, a message is returned with an XML string containing a list of all servers in the cloud, their clients, the client subscriptions, and other statistics.

4. What is in a message?

Each message has 2 strings by which it is directed – the subject and the type. The user must specify both for messages and subscriptions. In a subscription, each string may contain wildcard characters. The character “*” matches anything and the character “?” matches any one character. For example if a user subscribes to subject = * and type = *, all messages will be received.

The cMsg system includes some information within a message by default. The user can find the sender’s name, sender’s host, sending time, receiver’s name, receiver’s host, receiving time, cMsg version, domain name, whether the message is from sendAndGet, and whether the message is a response to a sendAndGet. There are also a number of user-settable fields in a message consisting of the subject, the type, a userInt which holds an integer, a user time, a text field of any length, and a byte array of any length. In the next release of cMsg, there will be a generic “payload” field which will hold any type of data. That is, the user will be able to add named fields to the message containing a string, any kind of int, a float, a double, another message, their arrays, and binary data. The payload fields are handled less efficiently than the built-in fields as they are converted to text and back again as needed; however, they provide a great deal of additional flexibility.

5. A Few Examples

Below we list some C++ code snippets demonstrating the simplicity and ease of the cMsg package. Note that the snippets below will compile and run when linked with the cMsg libraries, and that no IDL’s, stub generators, etc. are needed.

5.1. Sending a Message

```
#include <cMsg.hxx>

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

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

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

5.2. Receiving a message

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

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

//  do something else…
```

5.3. Synchronous messaging

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

6. cMsg Applications

Writing applications for DAQ systems becomes almost trivial with the cMsg package. Distributed with the software are some applications. The two of most general interest are the cMsgGateway, which cross-posts messages between domains, and the cMsgLogger, which logs messages to the screen, file, and/or database. Both are no more than a few hundred lines long. For the user to write an application like the logger simply requires a subscription to all messages and a callback which writes them to a file or database.

7. Performance

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

In figures 3 and 4 below, we show measurements of cMsg throughput employing both Java and C clients on Linux and vxWorks. In all cases the cMsg server was running on a 2.4 GHz quad-Opteron RHEL server, and all nodes had Gbit Ethernet interfaces connected to a Cisco Catalyst 4000 switch.

We identify two regimes: high message rate/small message size, or the “control” regime, and low message rate/large message size, the “DAQ” regime. The former is generally limited by CPU power on the client and server nodes, the latter by network bandwidth and resources required to service the network. The former is best understood from figure 3, the latter from figure 4.

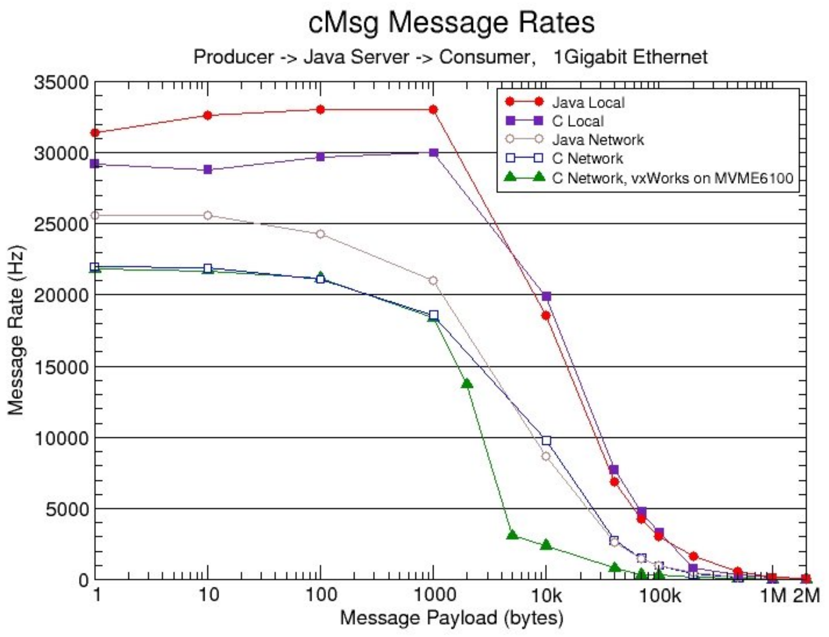

Figure 3. Message rate in Hz as a function of the message payload size in bytes. Messages have an overhead of 86 bytes.

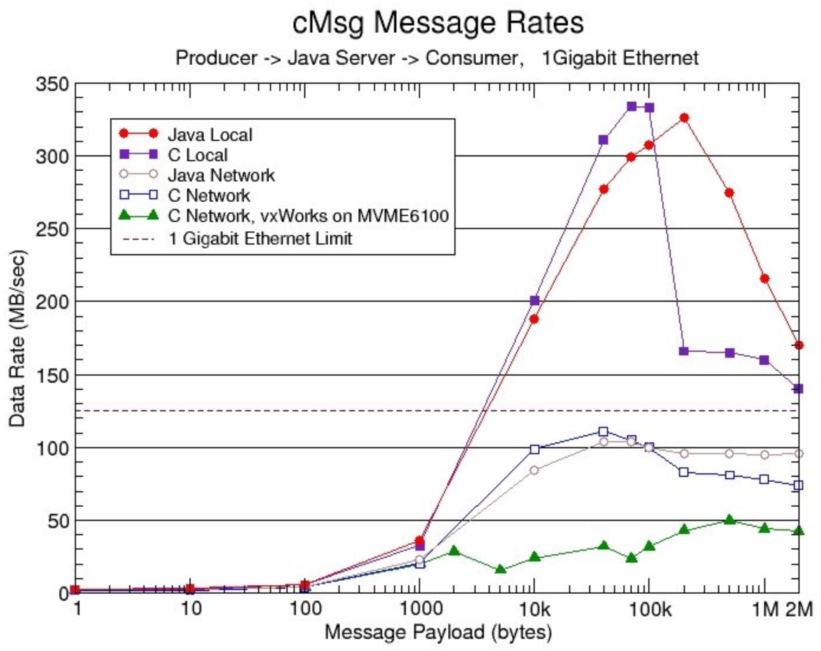

Figure 4. Data rate in MB/sec as a function of message payload size in bytes.

In figure 3 we plot message rate vs. message payload size (overhead per message is 86 bytes) for a number of different conditions. For the top two curves the producer and consumer ran on the same node as the server, so data did not move over a network. In the control regime the server handled over 33,000 messages per second with Java clients, and slightly less for C clients, somewhat surprising since one might expect C client performance to exceed Java client performance. These results place upper limits on server and client performance in our test setup, and are useful when interpreting later results.

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

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

Network bandwidth effects are most clearly seen in figure 4, where total data throughput is plotted vs. message size, and results become interesting above about 1 kByte message size.

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

In the network case the C rate peaks at about 110 MBytes/sec, or at almost 90% of the full Gbit bandwidth, but then falls off above about 100 kByte payload size, similar to the non-network case. We do not completely understand these falloffs at large payload size, but suspect they may disappear with careful tuning of the C code and network stack parameters.

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

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

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

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

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

We further had expected that the C performance would exceed Java performance, but this again was not the case. Despite careful tuning of the C code by an experienced C network programmer, the Java code out-performed the C code in the majority of our tests. To optimize Java performance, we specified certain flags to the Java Virtual Machine (JVM). Of all the JVM flags used, the "-server" flag is the most important for both client and server and optimizes operations for speed. The fact that the Java code runs at 80% of the Gbit bandwidth demonstrates that there is little left to be gained by further attempts to optimize.

9. Summary and Conclusions

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

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

The use of Java in cMsg greatly reduced our development time compared to C, and Java performance has proven to be excellent, generally exceeding C performance (although this may change with further tuning of the C components). Our results clearly demonstrate that Java is a serious contender for almost any DAQ or online requirement.

cMsg is available for download at ftp://ftp.jlab.org/pub/coda/cMsg. Give it a try.

References

[1] Heyes G, et al 1994 The cebaf on-line data acquisition system *Proceedings of the CHEP 1994 Conference*

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

[3] COSYLAB 2003 Abeans: application development framework for java *presented at the ICALEPC conference*

[4] DIM was developed at CERN and can be found at http://dim.web.cern.ch/dim