CMLOG: A Common Message Logging System

Jie Chen, Walt Akers, Matt Bickley, Danjin Wu and William Watson III
Control Software Group
Thomas Jefferson National Accelerator Facility
Newport News, Virginia 23606, U.S.A

Abstract

The **C**ommon **M**essage **Log**ging (CMLOG) system is an object-oriented and distributed system that not only allows applications and systems to log data (messages) of any type into a centralized database but also lets applications view incoming messages in real-time or retrieve stored data from the database according to selection rules. It consists of a concurrent Unix server that handles incoming logging or searching messages, a Motif browser that can view incoming messages in real-time or display stored data in the database, a client daemon that buffers and sends logging messages to the server, and libraries that can be used by applications to send data to or retrieve data from the database via the server. This paper presents the design and implementation of the CMLOG system meanwhile it will also address the issue of integration of CMLOG into existing control systems.

1 Introduction

A typical distributed message reporting and logging system as illustrated in Figure 1 informs operators with messages generated either from front end computers running real-time kernels or from applications on Unix hosts, stores the messages to a database, and allows retrieval of messages according to selection rules. In addition a logging client library is provided for applications to log messages to the database through the server. Most message logging systems currently available are influenced heavily by Unix standard error reporting syntax and are literally focused on string messages. The logging messages in these systems are usually in the form of either a predefined structure, which makes the whole system less adaptive to a new control environment, or a text string, which makes searching through the storage database less efficient. To store messages, some of these systems use simple ASCII files making fast searching almost impossible, or use commercial databases making the whole system less portable.

In contrast, the CMLOG system presented here uses logging messages in the form of a flexible C++ data object that holds multiple tagged data values of any type. This type of logging message allows applications and systems to log or report data of any type, and makes the CMLOG system easier to integrate into a control system. Moreover, the CMLOG system utilizes a well known database structure to allow fast searching, and employs a three-tier [1] network architecture to improve the scalability.

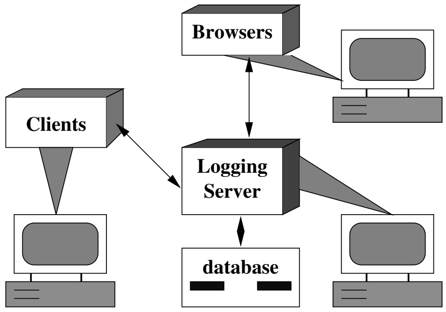

Figure 1: A typical distributed logging system.

2 Design and Implementation of CMLOG System

2.1 *Design and Analysis*

In comparison to a stand-alone error reporting system that writes error messages to local terminals or file systems of a host, a distributed logging system partitions the interactive GUI browsers, logging processes, the server, and persistent data storage among a number of otherwise independent machines in the network. At run-time, hosts running either real-time kernels or Unix operating systems (OSs) send logging messages to the server, while the interactive GUIs send request messages to and receive responses from the server. Although a distributed logging system offers better scalability and a better overview of a system, it is often significantly more difficult to design, implement, debug, optimize, and monitor than a stand-alone system. To handle the complexity of a distributed logging system, many topics (such as concurrency, connection management, message throttling, and resource management on real-time systems) have to be addressed. Object-oriented design and implementation techniques offer a variety of principles, methods, and tools that may help alleviate much of the complexity related to developing and configuring a distributed logging system.

2.1.1 Logging Messages and Network Protocol

The type of data used to log messages is crucial to the flexibility and performance of a logging system. A logging message usually consists of several predefined fields such as severity, status, user name, host name, and message string. Using a C or C++ data structure with several predefined fields as a log message prevents users from adding extra fields that may be important to different control systems. On the other hand concatenating all text strings converted from data of different types into a large text string as a long message indeed allows one to log data of arbitrary data types but severely hampers the operations of insertion and search of messages to and from the storage database, since only string comparisons can be performed. In order to let applications or systems log data of any type without sacrificing the performance, a message has to serve as a dynamic structure that contains multiple data fields, accessed by tagged values, of different types and sizes. The CMLOG system thus selects a C++ data type called *cdevData* [2] that was designed to be a repository for data of different types and sizes as the message type. The *cdevData* object uses either an integer tag or a character string tag to access internal data. There is a one to one correspondence between integer tags and string tags so either may be used to retrieve and insert data. Currently the *cdevData* object can hold any of type integer, pointer to a character string, char, short, ushort, uint, long, ulong, float, double, or time_val structure and arrays of these.

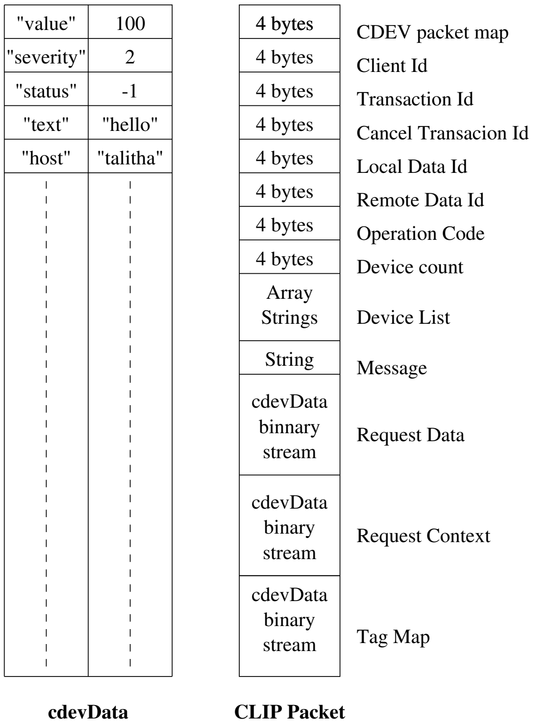

Figure 2: Illustration of *cdevData* and *CLIP* packet.

A network protocol is a set of rules that dictate how data and control information is exchanged between communication entities. The network protocol of CMLOG defines how messages of type *cdevData* are transferred between the server and logging clients, between the server and the browsers, and how the logging clients and browsers find where the server resides. The *CLIP* [3] protocol has been used to transfer *cdevData* between CDEV applications and CDEV servers. Using the *CLIP* protocol provides an efficient network transfer protocol for *cdevData* and makes integration of CMLOG system into the service layer of CDEV system much easier. Figure 2 illustrates the organization of the *cdevData* object and *CLIP* protocol.

2.1.2 Logging Clients

Logging clients in a logging system are applications that use an application program interface (API) to send messages to the server via established connections. Depending on the size of a control environment, there could be thousands of these applications running at the same time. Unfortunately, not all operating systems support thousands of network connections at the same time for a single network server. The CMLOG system introduces another layer of software between logging clients and the server, called a client daemon, to reduce the number of network connections on the server. On a given host there is only one client daemon that establishes network connection between the host and the server and sends all messages to the server collected from all logging clients on the host.

The logging clients and the client daemon can run on hosts running real-time kernels in addition to hosts running different flavors of Unix operating systems. A real-time kernel usually imposes tighter memory constrains, often provides no memory protection among kernel and user tasks, and offers different scheduling policies and global address space for kernel and tasks[4]. The logging client APIs and the client daemon thus have to be thread safe and consume little memory. Additionally, the logging client API should also provide callable routines for the Interrupt Service Routines (ISRs) which are common in real-time systems.

2.1.3 Browsers

The browser side of the logging system usually consists of a GUI interface that allows operators to monitor incoming logging messages in real-time and to retrieve messages from the database. A GUI interface of the CMLOG system lets operators select what and how to display the data fields of logging messages, since it has no knowledge about the fields until run-time. As a general logging system, CMLOG provides a browser API to enable programmers to develop customized display applications. Since the number of browsers running simultaneously is much smaller than the number of running client applications, browsers have direct connections to the server.

2.1.4 Storage Database

The database used to store message in a logging system has to be reliable and to allow fast insertion and search.

2.1.5 Server

The performance and reliability of the network server in a logging system is extremely important. The server receives all logging messages from all connected hosts, writes those messages to a database, and it has to handle requests from browsers to search the database. It is unacceptable if any network requests from browsers or clients are blocked for a long time while the server is waiting for disk I/O operations or is handling other network requests.

It is often a challenge to design and implement a robust and high performance network server. However the increased availability of advanced OS mechanisms (such as multi-threading), coupled with growing adoption of C++ and object-oriented methods, provides better understanding of the basic architectural choices for developing network servers.

- Connectionless or Connection-oriented protocols: *Connectionless protocols* (such as UDP, IP, CLNP) provide an unreliable, message-oriented service where each message may be routed independently. There is no guarantee that a particular message will arrive at its destination. In contrast, connection-oriented protocols (such as TCP) offer a reliable, sequenced, non-duplicated data delivery service for applications. A logging system cannot afford to lose messages sent by client applications or drop requests from browsers. It is thus obvious to choose connection-oriented protocols in the case of CMLOG. In particular the CLIP protocol is used on top of TCP.
- RPC or Lower-level IPC Mechanism: RPC is an attractive level of abstraction for developing network applications. They provide developers with a programming paradigm that closely resembles the familiar procedure calling conventions used in stand-alone applications. To implement a robust, portable and high performance network application such as the CMLOG system, however, it may be necessary to access lower-level IPC mechanisms such as sockets or TLI. Lower-level IPC mechanisms tend to be more efficient than RPC since they allow applications to omit some unnecessary functionalities and enable finer-grain control over communication behavior.
- Iterative vs. Concurrent Servers: An iterative server handles each client request in its entirely before servicing subsequent requests, which could be either blocked or ignored. In addition, iterative servers may also prevent clients from making progress while they are blocked awaiting their turn. Client-side blocking tends to complicate retransmission timeout calculations. This, in turn, results in excessive network traffic and may produce duplicate requests being received by a server. A concurrent server, on the other hand, handles multiple requests from clients simultaneously. It is well-suited for I/O bound and/or long-duration services that require a variable amount of time to execute. In a highly concurrent system such as CMLOG system, the server has to handle logging or searching messages from clients or browsers simultaneously. The iterative server scheme is therefore not suitable.

In summary, the CMLOG system is a connection oriented and low-level IPC based distributed logging system with a concurrent network server that offers fast logging and searching capabilities. Client applications log messages through a client daemon to the server. Browsers are directly connected to the server to search the database. Finally the architectural overview of the CMLOG system is presented in Figure 3.

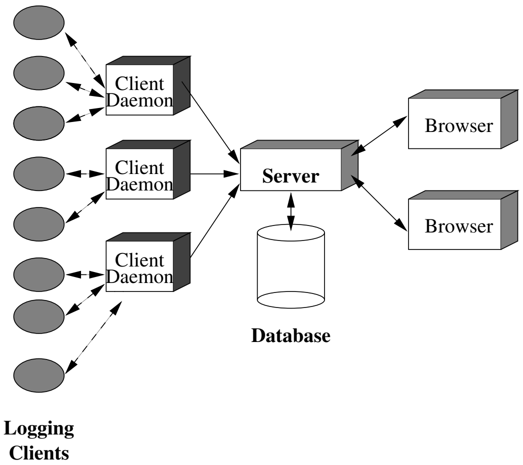

Figure 3: Architectural overview of the CMLOG system.

2.2 *Implementation*

Developing a distributed software system is difficult since it requires detailed knowledge of many concepts such as (1) network addressing and remote server identification, (2) creation, synchronization, and communication mechanisms for processes or threads, (3) presentation layer conversion techniques, and (4) task scheduling in real-time systems. Even though most operating systems offer various APIs for network IPC, using those APIs directly to implement the CMLOG system would lead to the following problems:

- Lack of Type-security: In most Unix and real-time systems a system call API identifies particular instances of I/O devices (such as files and sockets) using a common namespace consisting of unsigned integer I/O descriptors which are "weakly-typed" in the sense that disk file descriptors are not syntactically different from network connection descriptors. Therefore, it is easy to use the wrong descriptors in the wrong circumstances by accident.
- Non-Portability: It is difficult to write portable code that use OS IPC mechanisms since they are different among different operating systems, especially for real-time systems. This increases the complexity of developing and maintaining application source code.

Due to the efficiency and availability of C++, it makes sense to encapsulate the existing IPC and synchronization mechanisms for different operating systems within C++ classes and inheritance hierarchies. More explicitly, C++ classes handling task (thread) management and synchronization mechanisms for Unix and VxWorks have been developed, some small portion of the C++ classes from ACE [5] have also been tested on various platforms, including VxWorks. Developing the CMLOG system based upon these C++ classes helps improve software quality and portability.

2.2.1 Logging Client APIs and Client Daemon

The CMLOG client APIs are callable routines for applications. At run-time a client uses UDP messages to find out whether a client daemon is running on that host. If a client daemon is indeed present, the logging client uses a pipe to establish a communication channel. On Unix machines if there is no client daemon, a client daemon will be spawned. Each client is assigned a unique ID of type unsigned integer and has a logging context which contains fields (user name, host, etc.) that do not change. The client daemon uses a UDP broadcast to find a server on the subnet. The connected daemon then buffers all messages from clients on the same host and sends messages to the server. A non-connected daemon will periodically try to connect to the server while it writes all messages to the local console. Figure 4 presents an overview of clients and the client daemon of the CMLOG system.

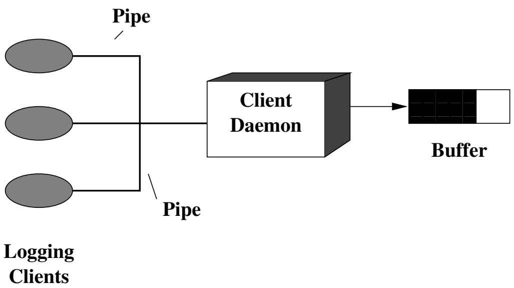

Figure 4: The logging clients and the client daemon.

All client APIs are handled by a single C++ class called *cmlogClient* which has three basic functions: (1) connect, (2) postData, and (3) disconnect. The following are sample C++ code for Unix and VxWorks systems using CMLOG.

```
#include <stdio.h>
#include <cmlogClient.h>
extern “C” int client_test (char* name);
#ifndef __vxworks
int main (int argc, char** argv)
#else
int client_test (char* name)
#endif
{
#ifndef __vxworks
    cmlogClient* client = new cmlogClient (argv[1]);
#else
    cmlogClient* client = cmlogClient::logClient (name);
#endif
    if (client→connect () == CMLOG_SUCCESS) {
      cdevData data;
      data.insert (“value”, 100);
      data.insert (“text”, “test 1 2 3”);
      data.insert (“severity”, -1);
      client→postData (data);
      client→disconnect ();
    }
}
```

Several routines similar to conventional Unix *printf* format with related C callable routines are also available. In addition, callable routines for ISRs are also provided.

2.2.2 Browser APIs and Motif browser

The CMLOG system provides a browser API that can be used to develop sophisticated applications that monitor logging messages in real-time and retrieve messages from the server. Applications using the API look for the server by UDP broadcast. Once the server is found, a TCP direct connection is established. The API allows applications to register callbacks with each request so that the applications do not have to wait for the requests to come back while processing other events (such as X events).

Additionally, the CMLOG system provides a sample implementation, called cmlog, using Motif and the browser API. The cmlog has the capability to let operators select what fields to display. It also lets operators adjust the display width for each field at run-time.

2.2.3 Server

The server in the CMLOG system deserves more attention. It is a concurrent server that handles logging and querying messages from clients and browsers while writing and reading data to and from a database. In the domain of network server, there is more than one way to achieve server concurrency.

- Multi-threaded implementation: Multi-threading are rapidly becoming available on most OS platforms [6]. A thread is an independent series of instructions executed within a single process address space. This address space may be shared with other executing threads. Threads are often characterized as “lightweight processes” since they maintain minimal state information, require less overhead to create and synchronize, and inter-communicate via shared memory rather than through IPC mechanisms. In a multi-processor machine a thread can bind to a particular processor so that parallel processing is possible. Even on a uni-processor machine a thread blocked on a system call will yield its cpu time to other threads, which results in higher network throughput and better response time. However, not all operating systems support multi-threading mechanisms.
- Multi-process implementation: In a Multi-process network server, each network request is handled by a Unix process which is either forked on demand or is pre-spawned into a poll at server creation time. Since a process is a kernel-scheduled entity, the concurrency is directly supported by the OS through round-robin scheduling. However, Unix processes are much more expensive to create and interprocess communications are more complicated than communication mechanisms among threads.
- Single-thread implementation: A concurrent server may also be designed to handle multiple requests si-

multaneously with a single-threaded process. Single threaded concurrent servers usually are implemented by explicitly time-slicing their attention to each request via techniques such as port demultiplexing (e.g. select or poll) and non-blocking I/O. In comparison with multi-threaded/multi-process servers, single-threaded concurrency servers can still be blocked if one request requires a long duration of service.

Network requests to the server of the CMLOG system are either to log messages or to query the database. The service durations for these requests range from very short for logging a single message to very long for retrieving a large number of messages in the database. It is therefore unwise to implement the server using a single-threaded implementation.

In order to make the CMLOG system portable, the server is implemented using either the multi-thread (using Pthread [7]) or multi-process method depending on the availability of Pthread on a platform. A set of parameters (such as default UDP port, number of threads (processes) to create, and shared memory ID) in a header file allows different sites to configure the server. In addition, the server reads a configuration file that contains more parameters which can be fine tuned for different environments without recompiling the code.

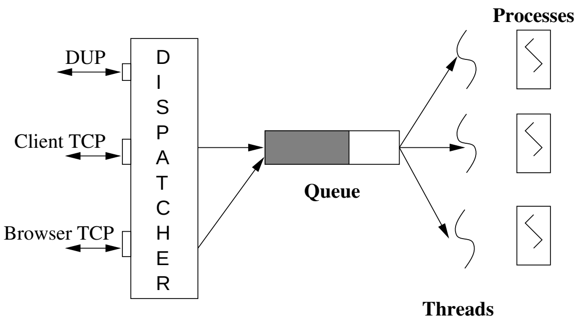

Figure 5: Run-time architecture of the server.

When the server starts up, it creates several ports to handle network requests. The first port is a UDP port which is well-known to clients and browsers. The others are TCP ports assigned by the operating system for TCP connections from client daemons and browsers. To find out where the server is on a subnet, client daemons and browsers send a UDP broadcast message to this UDP port. When the server receives the UDP broadcast message, it sends back information about what TCP ports the client daemon and browsers can use to establish TCP connections. Next, the server spawns the number of threads (processes) specified in the header file. These threads (processes) initially are asleep. When there is a network request, the master thread (process) wakes up one thread (process) to process the request. Once the request is serviced, the thread (process) goes back to sleep. This is very similar to the Master-Slave (Boss-Worker)[8] paradigm that is popular in parallel computation. Figure 5 demonstrates the run-time architecture of the server.

2.2.4 Database

The database contains multiple Unix files that contain time stamped logging messages of *cdevData* in binary form. Each file is indexed by time and is organized in a B+ tree structure. A database file is closed and a new file is created in a time cycle specified in the server configuration file.

3 Integration of CMLOG into a Control System

The flexibility of *cdevData* and easy customization of the server enables a CMLOG system to be integrated into a control environment easily. For example, one can keep the existing client logging API which then calls the client API of the CMLOG system. On systems such as EPICS [9] which allow error handlers to be installed to catch error messages, a simple routine implemented using CMLOG will send all messages to the server.

4 Concluding Remarks

The CMLOG system is a new distributed and general message logging system that enables applications or systems to log data of any type into a centralized database. It consists of a concurrent Unix server implemented in C++ using multiple threads or processes where applicable to improve network responsiveness and concurrency, a client daemon that buffers and sends all logging messages on a host to the server, a C++ client library that is used by applications to log messages, and a C++ browser library that supports callback mechanisms to let browsers handle other events and wait for responses from the server at the same time. Furthermore, the CMLOG system offers two CDEV services. One is *cmlogService* which is the CDEV service layer for the the client library. Another is *cmlogQService* which is the CDEV service for the browser library. These two services enable CDEV applications to log to and retrieve messages from the server in CDEV fashion. For example one can log messages in a CDEV application in the following way:

```
{
    cdevData data;
    data.insert ("severity", 2);
    data.insert ("status", -1);
    data.insert ("text", "Error happened");
    cdevRequestObject* obj = 0;
    obj = cdevRequestObject::attachPtr
            ("cmlog", "set msg");
    if (obj)
            obj→send (data, 0);
}
```

The CMLOG system currently serves as an error reporting system for the CDEV package and as an error logging system in the control system at Jefferson Lab. It has been tested on Hewlett-Packard machines running HPUX-9 and HPUX-10, Sun workstations running Solaris 2.5.x and Intel-based PCs running Linux 2.0.x. The client library and client daemon have also been tested on mv162, mv167, mv177 and mvme2604 running VxWorks 5.2(3). The server can handle up to 2000 and 200 logging messages per second from a Unix host and a mv162 machine respectively. It can send up to 1000 messages per second to a browser for a single query request. Finally the source code for CMLOG is available via anonymous ftp from ftp.cebaf.gov in pub/cdev.

5 Acknowledgment

Special thanks to Graham Heyes and David Abbott in the Data Acquisition Group of Jefferson Lab for running and debugging the CMLOG system on CODA [10]. Special thanks also go to Johannes van Zeijts in the control software group of Jefferson Lab for his valuable suggestions and code testing.

6 References

[1] Ban-Ari, M. "*Principles of Concurrent and Distributed Programming*". Englewood Cliffs NJ, Prentice-Hall. 1990.

[2] Jie Chen, Graham Heyes, Walt Akers, Danjin Wu and William Watson III, "CDEV: An Object-Oriented Class Library for Developing Device Control Applications", *Proceedings of ICALEPCS 1995*, p97.

[3] Walt Akers, William Watson III and Jie Chen, "The CDEV Linear Internet Protocol Definition". CDEV Reference Documentation.

[4] Phillip A. Laplante, "*Real-time Systems Design and Analysis, An Engineer's Handbook*". IEEE Press, 1993.

[5] D. C. Schmidt, "The ADAPTIVE Communication Environment: Object-oriented Network Programming Components for Developing Client/Server Applications". *Proceedings of the 11th Annual Sun Users Group Conference*, (San Jose, CA), SUG, Dec. 1993.

[6] J. Eykholt, et.al., "Beyond Multiprocessing... Multithreading and SunOS Kernel", in *Summer USENIX conference*, (San Antonio, Texas), June 1992

[7] IEEE, Inc., *Information Technology - Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) - Amendment 2: Threads Extension [C Language]*, IEEE Standard 1003.1c-1995, IEEE, New York, N,Y, Also ISO/IEC 9945-1:1990c.

[8] Steve Kleiman, Devang Shah, and Bart Smaalders, "*Programming with Threads*", Prentice Hall, Upper Saddle River, NJ, 1996.

[9] Leo R. Dalesio, et. al., "The Experimental Physics and Industrial Control System Architecture: Past, Present, and Future", *International Conference on Accelerator and Large Experimental Physics Control Systems*, Oct. 1993.

[10] William A. Watson III, Jie Chen, Graham Heyes, Edward Jastrzmbski, David R. Quarrie, "CODA: A Scalable, Distributed Data Acquisition System", IEEE Trans.Nuc.Sci., Vol. 41, p61.