

The Connection Manager Library

Greg Eisenhauer
eisen@cc.gatech.edu

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

September 17, 2015 – CM/EVPath Version 3.0

1 Introduction

Connection Manager is a library of communications routines which manage the complexity of systems with multiple communication links between heterogeneous machines. The library is designed to be used as an implementation basis for networks of agents communicating application-specific data. It contains support for establishing communication between agents, matching incoming messages with handlers, and assisting in the distribution of message format information to entities with which it is communicating. Connection Manager is a point-to-point messaging API contained within the broader EVPath library. This paper details the services and interfaces offered by CM.

2 Overall Description

The core purpose of Connection Manager is to ease the task of creating and operating networks of communicating entities over specialized and configurable data transport mechanisms. In particular, it is designed to abstract away the intricacies of those transports and to satisfy two conflicting goals: allowing uncustomized applications and libraries to transparently use specialized data transport mechanisms (such as raw ATM, InfiniBand, Reliable UDP), while still allowing knowledgeable application layers to configure transport particulars.

CM also directly supports heterogeneous applications by providing for binary data transmission between entities and locating the most appropriate handler for incoming data. To provide heterogeneity support, CM relies mostly upon FFS, a lower-level communications library that supports binary transmission of C-style data structures between heterogeneous machines. FFS is documented in the **FFS Reference Manual**, available at http://www.cc.gatech.edu/~eisen/FFS_manual.pdf. That document is not yet complete. It began its life as a manual for PBIO, FFS' predecessor, and has not yet been completely updated. Familiarity with the concepts and specifications used in FFS are necessary for understanding messaging in CM. Under normal circumstance, CM relies on FFS's internal mechanisms for distributing message format information. However, in circumstances

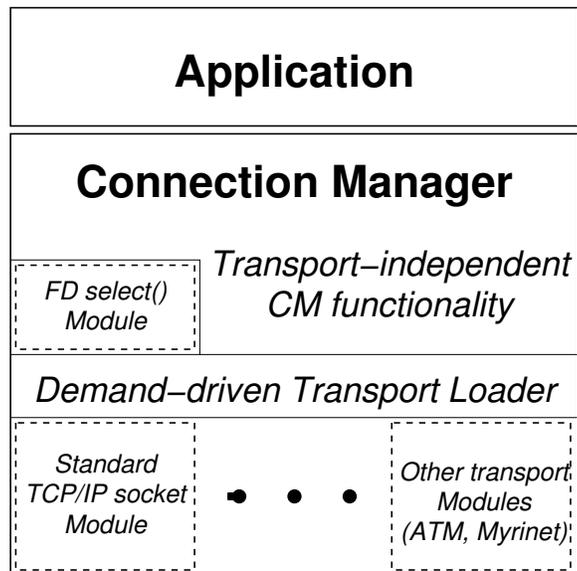


Figure 1: Structure of the Connection Manager.

where dependency upon FFS's external format server is impossible, such as for operation within the kernel, each CM application can act as its own format server.

In order to allow applications to transparently use a variety of data transport mechanisms, CM is structured so that individual transports are implemented with dynamically loadable modules as depicted in Figure 1. Applications can use transport mechanisms that might not have existed when the application as written and they are not burdened with the code and memory requirements of unused transports. In addition to the dynamically loadable transports, Figure 1 shows *FD select()* functionality separated out into a loadable module. This module provides control-flow support for transports which may be integrated into the OS file descriptor system. Its separation into a loadable module allows CM to be used as a communications manager even in situations where *FD select()* is not a viable control-flow mechanism, such as within the OS kernel.

In addition to allowing applications to transparently use different data transport mechanisms, CM is also designed to allow more aware applications to customize the behavior of those data transports, for example to specify bandwidth requirements, expected reliability or other transport-specific characteristics. Because CM does not have *a priori* knowledge of the specific transports that might be used by an application, it must have a mechanism through which it can pass virtually any type of parametric specification between applications and the selected data transport. This role is filled by *attribute lists*. An attribute is a *name/value* pair that specifies something about a connection or message. Lists of these attributes are used to specify to CM the characteristics of the connections it should make. For example, a standard TCP/IP connection might be specified by the attribute list:

```
{IP_HOST,"latte.cc.gatech.edu"},{IP_PORT,40767}
```

Attribute lists are used by CM for a variety of other purposes as well, in both the application-layer API and in communication with the data transport modules.

2.1 A “Hello, World!” Program

At this point it is useful to discuss Connection Manager in the context of a simple program. The program at the right is the receiving side of a basic “Hello, World!” program. CM provides a reactive programming style. That is, most programs are structured around the exchange of messages, utilizing CM’s ability to call application-specified handlers when messages of appropriate types arrive. This particular program does nothing except wait for the arrival of a “hello” message. When one arrives, the subroutine `msg_handler()` is called and it prints out the `string_field` element of the message.

The `typedef` at the beginning of the program declares the datatype of the message. The `FMField` declaration creates a FFS-style `FMFieldList` that is used to describe the message datatype to the Connection Manager. The subroutine `msg_handler()` has a prototype shared by all CM message handlers. It has parameters representing the calling Connection Manager, the connection on which the message was received, a pointer to the message data (as a `void*` pointer, and the `client_data` parameter that was given by the application when the handler was registered.

```
#include <stdio.h>
#include "atl.h"
#include "evpath.h"

typedef struct _msg {
    char *string_field;
} msg, *msg_ptr;

static FMField msg_field_list[] =
{
    {"string_field", "string", sizeof(char*), 0},
    {NULL, NULL, 0, 0}
};

static void
msg_handler(CManager cm, CMConnection conn, void *msg,
            void *client_data, attr_list attrs)
{
    printf("%s\n", ((msg_ptr)msg)->string_field);
}

int
main (int argc, char **argv)
{
    CManager cm;
    CMFormat format;
    attr_list contact_list;

    cm = CManager_create();
    CMlisten(cm);
    contact_list = CMget_contact_list(cm);
    printf("Contact list \"%s\"\n",
           attr_list_to_string(contact_list));
    format = CMregister_simple_format(cm, "hello",
                                     msg_field_list, sizeof(msg));
    CMregister_handler(format, msg_handler, NULL);
    CMrun_network(cm);
}
```

This program uses five basic CM functions:

`CManager_create()` – This call creates and initializes a `CManager` data structure. All CM applications must call this at least once. The `CManager` data structure provides a means of associating related communication channels. Essentially, the `CManager` is a control context in which network messages are handled.

`CMlisten(CManager cm)` – This call requests CM to listen for incoming network connections. What exactly this does depends upon what network transport layer is in use. In the default “sockets” transport, this will cause CM to create an internet socket on the current host, bind it to randomly chosen IP port `port_num` and begin listening for connections. Once `CMlisten()` has been called, other programs can initiate a connection with this program by calling `CMinitiate_conn()` discussed further below. The contact information for a CM

is encapsulated in an *attribute list*. The contents of the attribute list are *transport-specific*. Casual CM users should consider these attribute lists to be *opaque*. However, the goal of CM is that these attribute lists might provide the mechanism through which application-specific transport requirements, such as QOS specifications, can be communicated to the transport layer. Applications with detailed knowledge of the transport in use can use:

```
extern int CMlisten_specific (CManager cm, attr_list listen_info);
```

to pass specific attributes to the transport-based listen call. In particular, the “sockets” transport looks for an “IP_PORT” attribute to specify the port upon which it is to listen.

```
CMFormat CMregister_simple_format(CManager cm, char *format_name,
                                FMFieldList field_list, int struct_size);
```

– This routine provides the basic mechanism for making message formats known to CM. Registration is a prerequisite for handler registration or writing data of that format. The `field_list` parameter is a FFS field list specifying the structure of the message. Record format descriptors are described in the FFS documentation, but basically consist of a list of field descriptions, where each field description is a quadruple giving the field name, data type, size and offset within the record. Given this information (along with the format name and the overall size of the structure)¹, FFS can pack the record for transmission to other machines and can decode it despite differences in machine architecture or record layout.

`CMregister_simple_format()` can only be used to register message types that do not have internal substructures. For more complex data types, one should use

```
CMFormat CMregister_format(CManager cm, FMStructDescList format_list);
```

The `format_list` parameter is a list of `format_name/field_list` pairs as below:

```
typedef struct _FMformat_list {
    char *format_name;
    FMFieldList field_list;
    int struct_size;
    FMOptInfo *opt_info;
} FMStructDescRec, *FMStructDescList;
```

and is used to specify the representation of the top-level structure (entry 0 in the list) and any nested structures in the message. It should contain the transitive closure of all data types necessary to specify the message representation. The list is terminated with a `{NULL, NULL, 0, NULL}` value.

```
void CMregister_handler(CMFormat format, CMHandlerFunc handler,
                      void *client_data)
```

– `CMregister_handler()` binds the subroutine specified in its `handler` parameter to the arrival of messages of the type `format`. The profile of the handler function should be:

```
typedef void (*CMHandlerFunc) (CManager cm, CMConnection conn,
                              void *message, void *client_data, attr_list attrs);
```

¹Because of structure alignment considerations in the compiler, structure size cannot always be accurately inferred from information such as the size and offset of the last field. Therefore it should be specified explicitly with a `'sizeof()'` argument as shown in the examples.

```

#include <stdio.h>
#include "atl.h"
#include "evpath.h"
main(argc, argv)
int argc;
char **argv;
{
    CManager cm = CManager_create();
    CMConnection conn;
    attr_list contact_list;

    contact_list = attr_list_from_string(argv[1]);
    conn = CMinitialize_conn(cm, contact_list);
    CMConnection_close(conn);
    CManager_close(cm);
}

```

Figure 2: A simple client program.

The `client_data` parameter specified in the registration call is not interpreted by CM, but merely passed to the handler function when it is called. The `attrs` parameter points to an attribute list that includes any attributes specified in the `CMwrite_attr()` call made by the sender, as well as attributes that might have been added by the transport on the receiving side.

`void CMrun_network(CManager cm)` – `CMrun_network()` is one of the basic *network event* handling calls in CM. A CM network event is a basic communications occurrence, such as a connection request or message arrival. The routine `CMrun_network()` essentially handles network events until the CManager is shutdown. In this case, there is no `CManager_close()` call, so `CMrun_network()` will run forever.

A correspondingly simple client program that connects to this server is given in Figure 2. Unlike the sample server program, this client program doesn't really do anything useful. Assuming that the first argument to this program is the stringified attribute list printed out by the server program of Section 2.1, this client connects to the server, then immediately shuts down the connection (with `CMConnection_close(conn)`) and shuts down the CManager with `CManager_close(cm)`. The return value from `CMinitialize_conn()` is a value of type `CMConnection` and is a handle that can be used to write to the CM program on other end of that particular network connection.

These examples also introduce the two most important data types in CM, `CManager` and `CMConnection`. A `CManager` value essentially represents a communications context for a program. CM subroutines which operate on a communications context have a “CM” prefix in their name and take a `CManager` value as their first parameter. This paper will refer to a `CManager` value as “a CM.” Programs *can* create multiple CManagers and operations on those will be independent, but most programs will need just a single CM to support all messaging operations. In multi-threaded programs, only one thread should call `CMrun_network()` or the other network handling functions. `CMConnection` values are associated with a `CManager` and represent the endpoint of a bidirectional

communications link.

A CM acting as a simple client with a single communications connection will have only one `CMConnection`. A CM operating as a server with many connections will have many `CMConnection`'s, one for each client to which it is connected. CM communications links have a `CMConnection` on each end. CM's write subroutines for data operate on `CMConnection` values and send data across the communications link represented by their `CMConnection` parameter.

3 Sending Data

To actually make the program in Figure 2 useful, it has to send a message. Before a message can be sent, the message format must be registered with CM. This is done in the same manner as in the first sample program of Section 2.1 with the `CMregister_simple_format()` call. The `CMFormat` return value is then used in a call to `CMwrite()`.

```
extern int CMwrite (CMConnection conn, CMFormat format, void *data);
```

The body of the client program of Figure 2 then becomes:

```
{
    CManager cm = CManager_create();
    CMConnection conn;
    attr_list contact_list;
    CMFormat format;
    msg      message;

    contact_list = attr_list_from_string(argv[1]);
    conn = CMinitialize_conn(cm, contact_list);
    format = CMregister_simple_format(cm, "hello",
                                     msg_field_list, sizeof(msg));

    message.string_field = "Hello, World!";
    if(CMwrite(conn, format, (void*)&message) != 1) {
        printf("write failed\n");
    }
    CMConnection_close(conn);
    CManager_close(cm);
}
```

This program should be run with its first argument being the contact list string printed out by the example in Section 2.1.

3.1 Linking and Running

CM is built with some custom CMake macros that 1) build both a static library and a DLL (shared object for dynamic linking), and 2) create a .la for use with 'libtool'. libtool is closely associated with the autoconf/automake toolset and was designed to hide the complexity of building and using shared libraries behind a consistent, portable interface. At one point CM used libtool extensively, but we have now switched to CMake as a configuration tool, so libtool support is largely deprecated.

If you want to build a CM application with libtool, how this is done is best explained by the libtool documentation available from <http://www.gnu.org/software/libtool/libtool.html>. If libtool is utilized, using CM involves adding “-levpath” to the link line, preceeded if necessary by an appropriate “-Llibdirectory” flag. Libtool will locate any other libraries required by libtool.

At this point, it is more common to link CM applications without libtool. Depending upon the platform, it *may* be necessary to specify the transitive closure of all libraries that CM depends upon. A typical library spec might be “-levpath -latl -lffs”, again preceeded by an appropriate “-Llibdirectory” flag. (Some platforms, such as ancient Solaris, may also need “-lnsl -lsocket”.) However there are some subtleties that may cause difficulties. In particular, on many platforms, the linker will preferentially use the shared library if it is available. So, if both a .a and .so file are available in “libdirectory”, the .so form will be used with implications describe in the next subsection.

3.1.1 Using the shared library version

On many linux-flavored platforms, the -L flag is not the only flag that controls where shared libraries might be found. In particular, -L only controls the *link time* search path for shared libraries. Other flags, typically -R or -rpath, control the search path that will be used at *run time*. libtool would automatically do this for you, but you must do it manually if you don't use libtool. If you don't specify the appropriate run-time link path, the dynamic linker won't be able to locate the appropriate shared libraries and you'll get a run-time error. On most platforms, 'ldd' is a tool that lists the dynamic dependencies of an executable, including what the library search paths are and which libraries would be loaded if the program were run.

3.1.2 Using the traditional library version

Creating statically linked versions of programs can be useful for debugging. You can (generally) use the traditional library version (.a files) of CM if you specify the .a file directly. Additionally, some versions of ld accept flags that cause only the static versions of libraries to be used.

The principal caveat to using static libraries is that CM uses program-controlled dynamic linking (dlopen-style) to load its network transport layer. On some platforms, statically linked programs *cannot* use `dlopen()`. If CM is unable to load its transport layer, your program will exit with the error “Failed to initialize default transport. Exiting.”. You *may* be able to avoid this by linking only some libraries statically and letting others, particularly libc, be dynamic. libtool users can produce a completely statically linked executable because CM uses libtool's ltdl library. That library is capable of simulating dynamic linking in a statically linked environment (if all the dlls are known at link time.) See the libtool docs for how this works.

3.1.3 Running

Depending upon how they were linked, CM applications may require the environment variable **LD_LIBRARY_PATH** to be set at run-time. Using **LD_LIBRARY_PATH** can fix-up executables which do not have the correct run-time link paths built in with ld's `-rpath` flag.

3.1.4 Example Scripts

The tty sessions below demonstrate compiling, linking and running the programs a CoC RedHat box. The example of Section 2.1 is in the file "server.c" and the program of Section 3 is in "client.c". Note that the contact list specified to the client is protected from interpretation in the shell by quoting it.

```
scooby 1 > gcc -c -I/users/c/chaos/include server.c
scooby 2 > gcc -L/users/c/chaos/rhe5-64/lib -Wl,-rpath -Wl,/users/c/chaos/rhe5-64/lib -o server server.o -levpath -
scooby 3 > ./server
Contact list "AAIAAJTJ8o2lZQAAATkCmA4Fz4I="
Hello, World!
```

Figure 3: Compiling, linking and running the server.

```
scooby 1 > gcc -c -I/users/c/chaos/include client.c
scooby 2 > gcc -L/users/c/chaos/rhe5-64/lib -Wl,-rpath -Wl,/users/c/chaos/rhe5-64/lib -o client client.o -levpath -
scooby 3 > ./client "AAIAAJTJ8o2lZQAAATkCmA4Fz4I="
scooby 4 >
```

Figure 4: Compiling, linking and running the client.

3.2 Notes about Connection Manager

Handling of network contact information – In order to support many potential network transports and to allow their customization (with such things as Quality-of-Service parameters), CM uses *attribute lists*. As noted in Section 2, attribute is a *name/value* pair that specifies something about a connection or message. Lists of these attributes are used to specify to CM the characteristics of the connections it should make. In the case of TCP/IP socket connections, the attribute list required contains the same hostname/IP port pair that pretty much any network program would have used. Conceptually however, what is in the attribute list used to specify network contact information is of interest only to the CM transport layer making the connection. Non-specialized applications should treat the attribute lists as if they were opaque.

Attribute lists operations are supported by the "atl.h" include file and the "atl" library. The include file defines the `attr_list` data type. Most CM applications will need only two functions that operate on attribute lists. `attr_list_to_string()` converts an attribute list to string form and `attr_list_from_string()` does the reverse, parsing a string into an attribute list. Attribute lists can be freed with `free_attr_list()`.

```
extern char *attr_list_to_string(attr_list attrs);
extern attr_list attr_list_from_string(char * str);
extern void free_attr_list(attr_list list);
```

Note that the textual representation of attribute lists is a base64 encoding of a marshalled representation. In some circumstance it might be useful to see the details of the underlying attribute specification. The program `attr_dump`, included with the “atl” package takes a base64 attribute list as a parameter and prints out a (more) human-readable representation. For example:

```
scooby 10 > ~/bin/attr_dump "AAIAAJTJ8o2qZQAAATkCmG1+148="
Attribute list 0x7f8f08c00850, ref_count = 1
[
  { IP_PORT ('0x8df2c994'), Attr_Int4, 26026 }
  { IP_ADDR ('0x98023901'), Attr_Int4, -1881702803 }
]
```

Format handling is not “name” oriented – In some data communication systems, such as CM’s ancient predecessor DataExchange, message format registration and handling are heavily “name” oriented. Formats are named and perhaps only one handler for a particular message name could be registered. This was can be a simple arrangement, but it also seriously limits program adaptability and evolution. For example, if an server wanted to add a field to the request messages it accepted, it could not continue to service old clients by registering a handler for both the old and new message formats. CM has considerably more complex features that aid in program evolution, but in order to support them it is necessary to eliminate the idea that a format’s name is a unique way to reference it.

Eliminating the primacy of the format name frees CM to implement new features, but it does complicate the use of the library to some extent. For example, format registration incurs overhead, so it shouldn’t happen on every CMwrite. Instead, applications should save the `CMFormat` value that is returned by format registration for use in `CMwrite()`. Because data storage may be difficult in some situations, such as for libraries built on top of CM, there is a `CMlookup_format()` call that takes a `format_list` (not the format name) as a parameter and returns the `CMFormat` value that was created when the `format_list` was registered. This features relies on the fact that `format_lists` are typically statically allocated and their memory is not reused in the course of a CM application. *If you rely on this feature, you should ensure that all format lists used to register formats are unique. I.E. not dynamically allocated, free’d and potentially reused.*

Subformats in format registration – The example code above uses `CMregister_simple_format()` to make data formats known to CM, but that routine doesn’t support nested structures. To exchange more complex structures, `CMregister_format()` should be used and it’s `format_list` parameter should contain the transitive closure of all structures required to define the message. The order of the items in the list does not matter, EXCEPT that the zero’t item should be the top-level structure. For example, the doubly-nested structure `simple_rec` below can be registered using `CMregister_format` and `simple_format_list`.

```
typedef struct _complex_rec {
    double r;
    double i;
} complex, *complex_ptr;
```

```

typedef struct _nested_rec {
    complex item;
} nested, *nested_ptr;

typedef struct _simple_rec {
    int integer_field;
    nested nested_field;
} simple_rec, *simple_rec_ptr;

static FMField complex_field_list[] =
{
    {"r", "double", sizeof(double), IOOffset(complex_ptr, r)},
    {"i", "double", sizeof(double), IOOffset(complex_ptr, i)},
    {NULL, NULL, 0, 0}
};

static FMField nested_field_list[] =
{
    {"item", "complex", sizeof(complex), IOOffset(nested_ptr, item)},
    {NULL, NULL, 0, 0}
};

static FMField simple_field_list[] =
{
    {"integer_field", "integer",
     sizeof(int), IOOffset(simple_rec_ptr, integer_field)},
    {"nested_field", "nested",
     sizeof(nested), IOOffset(simple_rec_ptr, nested_field)},
    {NULL, NULL, 0, 0}
};

static CMFormatRec simple_format_list[] = {
    {"simple", simple_field_list},
    {"nested", nested_field_list},
    {"complex", complex_field_list},
    {NULL, NULL}
};

```

Adaptability features of format handling – Because CM is expected to be used in a potentially dynamic distributed environment, it does not assume that the incoming data will necessarily exactly match what handlers are registered. In fact, an application may have registered many handlers for a particular format name, each having different field and subformat lists. Thus, when an incoming message arrives CM tries to find the most appropriate handler for it. Matching is first done by the format name, and then by field lists. Any handler which requires fields not present in the incoming message is rejected as inappropriate. Among the remaining handlers, the one which matches the most fields in the incoming record is selected to handle the message. In the event of a tie, the format with fewest fields is selected.

Assuming that the format names match and ignoring data types for the moment, consider the following local formats with registered handlers: Handler 1 : Format has fields “a”, “b”
 Handler 2 : Format has fields “a”, “b”, “c”
 Handler 3 : Format has fields “a”, “b”, “c”, “d”

- An incoming message with fields “a”, “b”, “c” is passed to handler 2, matching all fields exactly.

- An incoming message with fields “a”, “b”, “c”, “d” is passed to handler 3.
- An incoming message with fields “a”, “b”, “d” is passed to handler 1, effectively discarding field “c”.
- An incoming message with fields “a”, “b”, “c”, “e” is passed to handler 2, effectively discarding field “d”.
- An incoming message with fields “a”, “c” is discarded because no handler matches.

In actual practice, most of these matching rules are unlikely to come into play. They just provide a format structure for CM to pick an appropriate handler while supporting the evolution of message formats in a system of communicating programs. For example, the rules mean that if a new field is added to an existing format, old clients can receive the new messages and with transparently convert them into the old format. At the same time, new servers can register handlers for both the old and new formats and have them called at the appropriate times.

no “read” call – Unlike some network packages, CM has no explicit network “read” call. Instead there is only the implicit read achieved through registering handlers. CM eliminates explicit read because it is appropriate only in the rarest of circumstances, and even then it is easily emulated while maintaining more generality than an actual explicit read. In particular, explicit read is only appropriate for a pure client with no ability to accept connections from others, only one current network connection, and absolute knowledge of the message type it is to receive next. Generally this only happens in client-side request-reply traffic. In this circumstance, it’s easy to emulate a read using CM’s condition variables (explained in more detail in Section 4.3). Instead of issuing an explicit read, as in the following code segment:

```
request_msg_t request;
reply_msg_t reply;
CMwrite(conn, request_format, &request);
CMread(conn, reply_format, &reply);
```

the program should register a handler for the reply message and use a CM condition variable to wait for the message. Using `CMCondition_wait()` has the advantage that messages of other types are properly handled while waiting for the reply. The following code substitutes for the write/read pair above, with an additional integer “condition” field in the request and reply formats:

```
request_msg_t request;
reply_msg_t reply;
int condition;
condition = CMCondition_get(cm, conn);
CMCondition_set_client_data(cm, condition, &reply);
request.condition = condition;
CMwrite(conn, request_format, &request);
CMCondition_wait(cm, condition);
```

In addition, a handler must be registered for the “reply” message format. This handler should be of this form:

```
extern void
Reply_handler(cm, conn, data, client_data)
```

```

CManager cm;
CMConnection conn;
void *data;
void *client_data;
{
    int condition = reply.condition;
    reply_msg_t *incoming_reply_msg = (reply_msg_t *) data;
    reply_msg_t *saved_reply_ptr;

    saved_reply_ptr = CMCondition_get_client_data(cm, condition);
    *saved_reply_ptr = *incoming_reply_msg;
    CMCondition_signal(cm, condition);
}

```

The request handler in the server should fill in the `reply.condition` field with the value from `request.condition`.

4 Control Flow

Unlike simple send-receive communications libraries, CM provides facilities that impact the flow of control in programs that use the library. While using CM does not imply a specific control model in the application and various control styles are possible, CM does need to know some basics, such as what thread library in use. Given that, it can infer other information, such as which thread is responsible for handling network traffic.

4.1 Thread Safety

CM/EVPath is a threaded and thread-safe environment. Generally CM protects itself and its own data structures with locking as appropriate. Currently CM builds with Pthread-based locking on all supported platforms. Depending upon the nature of CM/EVpath use, applications may need to protect their own data structures from concurrent access in handlers, etc.

4.2 Common Control Structures for CM Programs

The principal issue in considering control structures for CM programs is “what thread will service the network?”. If no one is servicing the network (receiving messages, accepting connections, etc.) distributed deadlock is possible when other programs try to communicate. Additionally, if multiple threads try to service the network, serious confusion may result inside CM. So for multi-threaded programs, exactly one thread should be responsible for servicing the network. This thread will also execute message handlers, so any long-running handler should be forked into a separate thread so that the original can go back to servicing the network.

“Servicing the network” is done by any of several mechanisms, including two explicit mechanisms that are normally used in non-threaded programs:

`void CMrun_network(CManager cm);` – A blocking call that causes the current thread to handle network requests until the `CManager` is shut down with `CManager_close()`.

`void CMpoll_network(CManager cm);` – A non-blocking call that handles any network requests that are pending at the time of the call and then returns control. Principally used in single-threaded programs that only occasionally service the network. Such programs must take care to call it regularly or risk delaying communicating programs.

In addition, several calls in CM *implicitly* service the network. Those calls are:

`void CMSleep(CManager cm, int secs);` Waits for `secs` seconds and then returns.

`void CMusleep(CManager cm, int usecs);` Waits for `usecs` microseconds and then returns.

`void CMCondition_wait(CManager cm, int cond);` Waits for the CMCondition `cond` to be signalled. Described in more detail in Section 4.3 below.

These calls can be used in threaded or non-threaded programs. In non-threaded programs, they always service the network while waiting. In threaded programs, their behavior depends upon what thread calls them. If they are called from a thread which does not normally service the network, they block using thread-based mechanisms until it is time for them to return. If they are called from the thread which normally services the network (such as when called from a handler), they service the network while waiting.

Figures 5 through 9 give some example control styles for threaded and non-threaded CM programs. Figure 9 shows the use of `CMfork_comm_thread()`, a useful utility function that forks a network handler thread if there is a kernel-level thread package in use. Otherwise it returns 0. This is just a convenience for programs that use threads packages, but it is yet more useful for libraries built on CM that might not have knowledge of the actual application’s use of threads.

4.3 Waiting for Conditions

As is mentioned in Section 3.2, CM provides a general mechanism for safely waiting for some condition, such as a response to an outgoing request, while continuing to handle incoming message. This is often necessary for libraries and programs which need to perform distributed operation in a “synchronous” manner. (That is, they don’t return until the operation is complete).

The CM facilities for waiting for some event to occur are similar to those associated with condition variables in threads libraries with the simplification that there are generally no mutex variables associated with these conditions and they can only be used once. The complete API for CM condition variables is give below:

```
int CMCondition_get(CManager cm, CMConnection conn);
int CMCondition_wait(CManager cm, int condition);
void CMCondition_signal(CManager cm, int condition);
void CMCondition_set_client_data(CManager cm, int condition, void* client_data);
void *CMCondition_get_client_data(CManager cm, int condition);
```

The typical use of these routines in a synchronous operation would be begin with the use of `CMcondition_get()` to “allocate” a condition variable. The ID of this condition variable (an integer for easy transmission) would then be sent along with the “request” message (the first part of the synchronous exchange). Then `CMCondition_wait()` is called to wait for the condition to

```

main() {
    CManager cm = CManager_create();
    /*
     * register formats and handlers
     */
    CMrun_network(cm); /* service network until shutdown */
}

```

Figure 5: A reactive server that runs forever (or until a handler calls `CManager_close()`).

```

main() {
    CManager cm = CManager_create();
    /*
     * register formats and handlers
     */
    CMSleep(cm, 600); /* service network for 600 seconds */
    CManager_close(cm);
}

```

Figure 6: A reactive server that runs for a specified time.

```

main() {
    CManager cm = CManager_create();
    /*
     * register formats and handlers
     */
    while (!done) {
        /* do some work here */
        Cmpoll_network(cm);
    }
    CManager_close(cm);
}

```

Figure 7: A reactive server that does work in between handling the network.

```

main() {
    CManager cm = CManager_create();

    /*
     * register formats and handlers
     */

    /* fork worker threads */
    (void) pthread_create(&worker_thread, NULL, worker_task, (void *)data);
    (void) pthread_create(&worker_thread2, NULL, worker_task2, (void *)data2);
    CMrun_network(cm); /* service network until shutdown */
    CManager_close(cm);
}

```

Figure 8: A threaded program where the main program handles the network.

```

main() {
    CManager cm = CManager_create();

    /*
     * register formats and handlers
     */
    CMfork_comm_thread(cm); /* fork network handler thread */
        /* fork worker threads */
    (void) pthread_create(&worker_thread, NULL, worker_task, (void *)data);
    (void) pthread_create(&worker_thread2, NULL, worker_task2, (void *)data2);
    pthread_join(work_thread, &status);
    pthread_join(work_thread2, &status); /* wait for threads to exit */
    CManager_close(cm);
}

```

Figure 9: A threaded program that forks a network handler thread while the main program does other things.

be signaled. While waiting, CM will perform whatever action is necessary in order to continue handling requests from the network. In order to complete the synchronous operation, a handler should have been registered for the “result” message and that message should contain the ID of the condition upon which the initiator is waiting. The handler’s role is to store the results somewhere and use `CMCondition_signal()` to signal the occurrence of the condition. The routines `CMCondition_set_client_data()` and `CMCondition_get_client_data()` are provided to allow an arbitrary client address to be associated with the condition variable. This simplifies communication between the initiating and the handling routines because the initiator can setup a results area and associate its address with the condition, where it can be retrieved by the handler routine. The code in Section 3.2 showing how a `read()` operation can be replaced with a `CMCondition_wait()` provides a concrete example of how CMConditions might be used.

There are two exceptional conditions worthy of discussion. The first concerns the situation where the far end of the synchronous operation dies between the time the request is sent and the time the response is received. CM cannot handle this situation totally transparently because more than one host may actually be involved in completing a remote operation. However, if the `CMConnection` parameter in the `CMCondition_get()` is set to a non-NULL value, the `CMCondition_wait()` will terminate if the indicated connection is closed. In this situation, `CMCondition_wait()` will return 0 rather than its normal 1 return.

The second exceptional condition concerns the side effects that might be encountered if somehow a condition is never signaled or completed via connection closure. Essentially, the results and side-effects in this circumstance depend a great deal upon the application control structure and any underlying thread library. The possibilities range from a thread that is blocked forever to a stack leak.² However, none of the possible results are likely to be desirable and robust applications should restrict their use of CM conditions to circumstances where waiting periods are relatively short and response is assured. In other circumstances, behavior is unpredictable.

²A stack leak is a permanent loss of stack space, such as might occur if a subroutine always pushed more items on the stack than it removed. In this case, it might be caused by an event handler subroutine calling itself recursively but never returning.

5 Miscellaneous Features

The previous sections have covered the most basic of CM functionality. This section wraps up some loose ends and covers smaller topics that do not fit well elsewhere.

5.1 Data Management

Generally speaking, incoming message data in CM is guaranteed to remain valid only for the duration of the execution of the handler function. Thereafter, the memory where the data is stored is subject to being overwritten or `free()`'d. Application which need to store all or part of the incoming message beyond the lifetime of the handler should copy the data into application-managed memory. Alternatively, the application can use the routine `CMtake_buffer()` to inform CM that it is taking control of the buffer containing the incoming message data.

```
extern void CMtake_buffer(CManager cm, void *data);
extern void CMreturn_buffer(CManager cm, void *data);
```

The `data` parameter is the address of the incoming data (as provided to the handler). After `CMtake_buffer()` has been called during processing of a record, the CM library allocates itself a new buffer for holding input data and stops referencing the old buffer. The application is then responsible for eventually passing the buffer to `CMreturn_buffer()` when it is no longer of use.

5.2 Useful Variants of Standard Routines

In addition to the basic routines described in prior sections, there are a couple of variations that are useful in some circumstances.

```
CMConnection CMget_conn(CManager cm, attr_list contact_list)
```

`CMget_conn()` behaves like `CMinitiate_conn()` except that it first checks to see if an existing connection matches the contact list. If so, it increments the reference count of that connection and returns it. Otherwise it initiates a new connection.

```
int CMwrite_attr(CMConnection conn, CMFormat format, void *data, attr_list attrs)
```

`CMwrite_attr()` allows an attribute list to be specified along with the write request. While no current transports interpret per-write attributes, this call could be used to specify QoS parameters, priorities, deadlines, etc. `CMwrite()` is equivalent to `CMwrite_attr()` with `NULL` passed for the `attrs` parameter.

5.3 More Rarely Used Routines

There are also some rarely used routines that fill special needs. In particular:

```
int CMcontact_self_check(CManager cm, attr_list attrs)
```

Generally, it is bad form for an application to try to initiate a connection to itself.³ However, contact lists are semi-opaque abstractions with properties that vary with various transports, so checking to see if a contact list denotes yourself is not necessarily straightforward. This is where `CMcontact_self_check()` comes in. It interacts with the CM transports to answer the question “Does this contact list point to me”. It returns 1 if yes, 0 if no.

```
void CMregister_close_handler(CMConnection conn, CMcloseHandlerFunc func,
                             void *client_data)
```

This call provides a mechanism through which an application can be notified when one of its connections closes, either through the application calling `CMConnection_Close()` or through a failure or shutdown of the network-layer connection.

```
CMregister_non_CM_message_handler((int header, CMNonCMHandler handler)
```

It is not unusual for messaging schemes to use a “magic number” as the first element of their message in order that they are recognized on arrival. In CORBA’s IIOP, for example, all messages begin with the string “GIOP”. In CM, messages begin with the 4-byte integer `0x434d440` (the string “CMD\0” expressed as an integer). This arrives as `0x00444d43` when the native byte orders of the parties differs. `CMregister_non_CM_message_handler()` provides a mechanism through which CM can be extended to directly accept and process IIOP and other messages. The handler routine is passed the `CMConnection` upon which the message has arrived and the 4-byte integer with which the header has been recognized. The remainder of the message must be read directly from the connection. The prototype of the handler is:

```
typedef void (*CMNonCMHandler) (CMConnection conn, int header);
```

5.4 Asynchronous Task Support

CM contains the following support for asynchronous tasks:

```
CMTaskHandle CMadd_periodic_task (CManager cm, int period_sec, int period_usec,
                                  CMPollFunc func, void *client_data);
```

This call schedules a function to be called repeatedly, at a regular period. The period is specified in seconds and microseconds. The function (whose profile is specified below) is passed only the `CManager` value and the `client_data` value. The periodic task can be cancelled with the `CMremove_task()` call below.

```
CMTaskHandle CMadd_delayed_task (CManager cm, int secs, int usec,
                                  CMPollFunc func, void *client_data);
```

This call schedules a function to be called one time only, after a particular period of time has elapsed. The period is specified in seconds and microseconds. Otherwise the behavior is like `CMadd_periodic_task()`.

```
void CMadd_poll (CManager cm, CMPollFunc func, void *client_data);
```

This call adds a function that will be called “occasionally.” In particular, it will be called after every message arrival and every timer expiration. The function cannot be cancelled.

³Ever use a phone to call its own number? Sometimes if you dial and hang up quickly enough you can call your own phone. Not always though, it depends upon what phone company hardware is involved. Initiating a network connection to yourself is like that. Sometimes it works, othertimes it deadlocks, depending upon the OS and transport involved.

All calls specifying a period are best-effort and are only as accurate as the underlying infrastructure. Generally timing functions somehow map into a `select()` call with a particular timeout.

```
typedef void (*CMPollFunc)(CManager cm, void *client_data);  
void CMremove_task (CMTaskHandle handle);
```

5.5 Shutdown

```
void CMConnection_close(CMConnection conn)
```

This call is used to shut down a connection. This call normally disables writing and incoming messages on the connection as well as closing the underlying network link. However, because `CMget_conn()` can return an existing connection (as opposed to initiating a new connection), CM maintains a reference count on connections. `CMConnection_close()` decrements this reference count and only has its final effect only when the reference count for the connection reaches zero. In general, the number of `CMConnection_close()` calls should match the number of `CMinitiate_conn()` and `CMget_conn()` calls. However, if connection was accepted implicitly through a `CMlisten()` call, it can still be shutdown with `CMConnection_close()`. `CMConnection_close()` will be called implicitly if there is a permanent write error on the connection or if CM detects that the network-layer link is terminated. The application is also free to call `CMConnection_close()` within a handler using the `CMConnection` value passed in to the handler function.

```
void CMConnection_add_reference(CMConnection conn)
```

This call increments the reference count of a connection. This is useful if the application passes `CMConnection` values to other contexts. Then each context can call `CMConnection_close()` and only the final close will have effect.

```
void CManager_close(CManager cm)
```

This call shuts down an entire CM. It closes all connections, and terminates any network handler thread forked with `CMfork_comm_thread()`. It also causes any `CMrun_network()` call to return to the caller.

5.6 Debugging

CM contains relatively a extensive tracing facility that dumps debugging output to standard output. This tracing facility is turned on by setting shell environment variables prior to running the program. The following environment variables are understood:

CMDataVerbose – Causes CM to print out the contents of every message that is sent or received.

CMConnectionVerbose – Prints information about connection initiation and acceptance.

CMControlVerbose – Prints information about control flow, condition waits, periodic task scheduling, etc.

CMTransportVerbose – Turns on tracing of transport-level happenings.

CMLowLevelVerbose – Traces low-level read/write and locking behavior. (Probably only useful for CM developers.)

CMVerbose – Turns on *all* CM*Verbose traces except CMLowLevelVerbose.

CMDumpSize – Controls the maximum number of length of the message contents output that CMDataVerbose displays. Default value is 256.

6 Writing a CM Transport DLL

WAY outside the scope of this paper. Contact the author.

References