Building Communication Paradigms with the CM-5 Active Message layer (CMAM)

Thorsten von Eicken
David E. Culler

Computer Science Division — EECS
University of California, Berkeley
Berkeley, CA 94720
(510) 642-8299

July 3, 1992

1 Introduction

The CM-5 Active Message layer (*CMAM*) provides a simple, general purpose communication primitive as a thin veneer over the raw hardware. It is intended to serve as a substrate for building libraries that provide higher-level communication abstractions and for generating communication code from a parallel-language compiler, rather than for direct use by programmers. *CMAM* is currently in use at UC Berkeley under the Split-C and Id compilers, as well as in a variety of libraries. (Of course, some people insist on writing in *CMAM* directly. No accounting for taste.) This document describes the basic concepts embodied in *CMAM* and illustrates how various abstractions can be constructed on this substrate. For specifics on the interface, consult the manual pages (appendix A). Experience with the CM-5 is assumed.

The basic communication primitive provided by *CMAM* is an Active Message: a message with an associated small amount of computation (in the form of a handler) at the receiving end. In *CMAM*, the first word of a message points to the handler for that message. On message arrival, the computation on the node is interrupted and the handler is executed. The role of the handler is to get the message out of the network, either by integrating it into the ongoing computation or by replying, in the case of a remote request. The buffering and scheduling provided in *CMAM* are extremely primitive and thereby fast. The only buffering is that involved in actual transport and the only scheduling is that required to activate the handler. This is sufficient for many applications. More general buffering and scheduling can be easily constructed in layers above *CMAM*. This minimalist approach avoids paying a performance penalty for unneeded funtionality. The power of Active Messages comes from the ability to customize message formatters and handlers, and from the simplicity (hence efficiency) of the implementation. Currently a five word *CMAM* packet can be sent in 1.4us and dispatched to an arbitrary C function on the receiving end in 1.6us.

A few aspects of active messages and the current implementation on the CM-5 are worth noting at the outset.

- The role of message handlers is to service the network, i.e., to reply to requests or to store the data of incoming messages into memory in a reasonable way. Message handlers are not intended to perform computation as such, but are not prevented from doing some reasonable amount of work. In this sense the active message model is fundamentally different from message-driven models of computation (e.g., dataflow, concurrent smalltalk, concurrent agregates, etc.). In some cases, it is cheaper to process the data on-the-fly than to copy it to a buffer and handlers are often specialized in this way. It is generally best if all storage required to handle a message is preallocated.

- The current *CMAM* implementation polls the network in order to accept messages and execute the appropriate handlers. This is buried within the various *CMAM* operations, so it is generally invisible. However, the `CMAM_wait` primitive, or an equivalent poll loop, should be used for busy waiting. Using interrupts to receive messages is more natural to active messages, but the costs on the CM-5 are rather steep. The interval between polls increases the latency of remote requests which affects the prefetching distance or multithreading required. Failure to service the network for long periods may cause messages to back-up into the routers. It is anticipated that later versions will utilize interrupts.

- The size of message packets on the CM-5 is limited to five words. *CMAM* provides variable length active messages as the primary interface, but also exposes an interface that reflects this physical limit. The CM-5 network allows packets to be re-ordered along a point-to-point connection, so there is some overhead to forming multipacket messages. *CMAM* provides *communication segments* to facilitate bulk transfer and all-to-all personalized communication. When a certain amount of data is deposited into the segment the handler is activated.

- Deadlock avoidance is a key aspect of the design of the *CMAM* layer, but still needs to be understood when constructing communication layers on *CMAM*, especially when handlers reply to remote requests as in a shared memory model, for example. Networks are usually "deadlock-free" as long as processors absorb messages. One-way communication is straight-forward; the sender blocks if the network cannot accept packets, but continues to receive and handle incoming packets. Two-way communication is trickier since handling an incoming request involves sending a reply. Accepting requests while replies are blocked is likely to lead to stack overflow. In general, requests and replies must have different priority levels, so this distinction is explicit in the *CMAM* interface. On the CM-5 the priorities are established by the usage of the two data networks; one is reserved for requests and the other one for replies. The processor must always accept packets from the reply network, but can stop listening to the request network while servicing a request, even if the request blocks. (The implementation of active messages on other architectures achieves this deadlock avoidance through other means.) For one-way communication either network can be used and the *CMAM* interface provides calls to inject packets into either network.

The core of *CMAM* is in three parts. The message sending and reception primitives are hand-coded in assembler. Header files contains ANSI-C function prototypes and GCC inline functions to poll the network and to format simple requests. Several communication abstractions, including shared memory and traditional message passing, are provided as formatters and handlers are written in C. As the examples below show, the *CMAM* user is permitted, even encouraged, to extend these with specialized format and handler functions which can be far more effective than the standard ones.

The current header files assume that GCC is being used. Converting the prototypes and using macros instead of inline functions is, in principle, possible. Note that there is no special compiler for the CM-5, any Sun-4 compiler can be used. Only the linker is special and produces two executables, one for the CN (control node, scalar, or front-end) and one for the PNs (processing nodes).

In general, a communication operation built on top of *CMAM* includes a formatter and a handler. Usually, the formatter is trivial and simply uses *CMAM* to call the handler function on a remote processor. For two-way communications, the request handler will use `CMAM_reply` to call a function on the requesting processor. This is illustrated by example in the following sections.

2 Hello world

This section describes a trivial program which pings all processors using *CMAM* and prints a hello string. A minimal *CMAM* program consists of three files: the program running each of the nodes, the program running on the host and a C-prototype file to link the two together. This is identical to programs written using the CMMD message passing library and is described in further detail in the "CMMD User's Guide".

2.1 Host program

Figure 1 shows the host code for our first program. When execution begins `main` is called on the host (as in any standard C program) while all processing nodes (PNs) enter a dispatch loop. `Main` first enables *CMAM* and the *CMAM* synchronization barrier and then calls `do_pings` on the PNs. `Do_pings` really is a stub generated from the prototype file in Figure 2 and causes the dispatcher on each PN to call `CMPE_do_pings` of the node program. The argument to `do_pings` is the handler to be called by PN 0 to return the total execution time. The call to `do_pings` is asynchronous, i.e. the host program continues to execute without waiting for `do_pings` to complete. After starting `do_pings`, the host program calls `CMAM_wait` to wait for the `done` flag. `CMAM_wait(&flag, value)` busy-waits for the `flag` to reach `value`. While busy-waiting, it polls the network (otherwise nothing would ever happen) and before returning, it resets `flag` by subtracting `value`. After printing a "Hello World" with the timing information, `main` disables *CMAM* and terminates. `CMAM_disable` waits for all PNs to return to the idle loop before disabling *CMAM*. The call to `exit` not only terminates the host programs but also kills all PN programs.

Observe that *CMAM* operations work between the host and the nodes as they do between nodes. However, there are some underlying differences. The nodes all have identical code address spaces. Thus, the local address of a handler is also its remote address. This uniformity does not hold for the host, since it has a completely different address space. This is why the host handler address is passed as an argument to the node code. Secondly, the performance of node-to-host operations is much worse than other operations because the node operating system is involved.

`CMAM_barrier` is enabled separately from the rest of *CMAM* because it changes the host participation in global synchronization operations vis a vis CMMD. Under CMMD the host participates, under *CMAM* it does not. Mixing CMMD and *CMAM* is somewhat subtle and

```
#include <stdio.h>
#include <cmam/cmam.h>

/* receive result from PNs */
static volatile int done = 0;                                /* flag set when PN part finished */
static volatile double time = 0;                             /* time taken by PN part */
static void set_time(double t) { time = t; done++; }         /* handler for PN 0 to call */

/* main program */
void main(int argc, char **argv)
{
        CMAM_enable();                                        /* enable CMAM */
        CMAM_enable_barrier();                                /* enable barrier separately */

        do_pings(set_time);                                   /* run PN code */
        CMAM_wait(&done, 1);                                  /* wait for PN code to complete */

        fprintf(stderr, "Hello world from %d nodes.  Pings took %.1fus each.\n",
                CMAM_partition_size, (time*1E6)/CMAM_partition_size);

        CMAM_disable();                                       /* sync all PNs and disable CMAM */
        exit(0);
}
```

Figure 1: *Host program for the "Hello World" example.* File `ping_host.c`.

discussed further in Section 6.

```
void do_pings(void (*set_time)());
```

Figure 2: *Stub prototypes for the "Hello World" example.* File `ping.proto`.

A prototype file, shown in Figure 2, connects the host call to the node routine, shown in Figure 3. Note, however, that the routine called on the node must have a `CMPE_` prefix, *i.e.*, calling `do_pings` on the host goes through a stub and invokes `CMPE_do_pings` on all the nodes. The node program shows the three parts of a two-way communication operation: `ping`, `pong`, and `poof`. Ping uses the machine specific `CMAM_4` call to invoke `pong` on `dest_node` passing it the requesting node number and the address of a flag on the requesting node. Then `ping` busy waits on this flag. The *CMAM* call invokes the function specified by its second argument on the node specified by its first argument with the remaining arguments as input. The CMNA timer registers are used in the example to time this loop. These timers provide elasped time in processor cycles (33MHz) including timesharing gaps.

The `pong` function is invoked by *CMAM* on the remote node with the return node and flag address as arguments. It simply invokes `poof` on the requesting processor using `CMAM_reply`. The reply version must be used within request handlers. Response handlers are not permitted to send additional messages. `poof` simply increments the flag specified in its argument. This is the flag that ping is busy waiting on, so ping will return from the `CMAM_wait` call after poof is invoked. The top level node routine exercises the ping-pong-poof from node zero to each of the other nodes. It then delivers the aggregate time to the host. Meanwhile, all the other nodes are waiting in the `CMAM_barrier`. Nodes continue to services requests while in the barrier. After all nodes have execute the `CMAM_barrier` call, each is allowed to proceed. In this case, they all complete and return to the dispatch loop, whereupon the program terminates.

```
#include <cmam/cmam.h>
#include <cm/timers.h>                                   /* access to hardware timer */
static double time(int t)                                  /* turn tick into seconds */
{ CM_TIME ct = {0, t}; return CM_ni_time_in_sec(ct); }

void CMPE_do_pings(void (*set_time)())
{
        int me, dest;
        double time;
        static double ping(int dest_node);

        me = CMNA_self_address;

        if(me == 0) {
                time = 0.0;
                for(dest=CMNA_partition_size-1; dest≥0; dest--) {
                        time += ping(dest);
                }
                CMAM_host_4(set_time, time);
        }

        CMAM_barrier(0);
}

static void poof(volatile int *flag) { (*flag)++; }

static void pong(int return_node, volatile int *return_flag)
{ CMAM_reply_4(return_node, poof, return_flag); }

static double ping(int dest_node)
{
        int t;
        volatile int done = 0;

        t = *(volatile int *)time_reg;
        CMAM_4(dest_node, pong, CMNA_self_address, &done);
        CMAM_wait(&done, 1);
        t = *(volatile int *)time_reg - t;
        return time(t);
}
```

Figure 3: *Node program for the "Hello World" example.* File `ping.c`.

The `Makefile` for the "Hello World" program is shown in Figure 4. The first two lines define the location of the *CMAM* libraries and *include* files. This may vary depending on the installation. *CMAM* requires the use of the GNU C compiler, preferably version 2.0 or later. If you use GCC 2.xx, you must load the gcc library with both the host executable and the node executable, and the C compiler flags must include `-gstabs` to allow `pndbx` (the CM-5 node debugger) to read the debugging information generated by GCC. The additional `-Dpe_obj` and `-DPE_CODE` flags are required for compiling the node program. The last rule in the makefile converts the prototype in `ping.proto` into a stub to be loaded with the host program.

```
CMAM_LIB=/usr/cm5/local/lib
CMAM_INC=/usr/cm5/local/include/cmam

CC=gcc-2
CFLAGS=-gstabs -O2 -I/usr/cm5/include -I$(CMAM_INC) -DCM5
PNFLAGS=$(CFLAGS) -Dpe_obj -DPE_CODE
GCC2LIB=/usr/local/lib/gcc-lib/libgcc.a

CP_LIBS=-L$(CMAM_LIB) -lcmam -lcmna_sp -lm $(GCC2LIB)
PN_LIBS=-L$(CMAM_LIB) -lcmam_pe -lcmna_pe -lm $(GCC2LIB)

PGM_CP=ping_host.o ping_proto.o
PGM_PN=ping.o

ping: $(PGM_CP) $(PGM_PN)
	cmld -o ping $(PGM_CP) $(CP_LIBS) -pe $(PGM_PN) $(PN_LIBS)

.c.o:
	$(CC) $(PNFLAGS) -c $*.c

ping_host.o: ping_host.c
	$(CC) $(CFLAGS) -c $*.c
ping.o: ping.c

ping_proto.o: ping.proto
	sp-pe-stubs <ping.proto >ping_proto.c
	$(CC) $(CFLAGS) -c ping_proto.c
```

Figure 4: *Makefile for the "Hello World" example.* File `Makefile`.

3 Shared Memory

This section shows how shared memory primitives can be built on *CMAM*This is essentially like the ping-pong example in the previous section. The implementation of global read and write function is shown in Figure 5. A counter `sm_flag` is used to keep track of the number of outstanding shared memory operations. For a simple shared memory model this will be one or zero. The `release` routine waits until all previous shared memory operations have completed, *i.e.*, until the counter reaches zero. Observe that it calls `CMAM_poll` to ensure that incoming messages are received and handled.

The read operation has three parts. `read_i` waits for previous requests to complete and then sends an Active Messageto read the desired location on the remote processor. The counter is incremented to reflect that a request is outstanding. The handler reads the location and replies to the originating processor. The response handler deposits data in a buffer and decrements the counter, so the original `read_i` observes the completion and can return the data.

The write operation similarily has three parts in order to provide sequential consistency. The request contains the address and the data. The handler deposits the data in the desired location on the remote node and replies with an acknowedgement. The `write_i` does not wait for the acknowledgement, but it does wait to ensure that all previous acknowledgements have been received before initating the write. Thus, there can be at most one outstanding write and the writes from each processor occur in the order that they are issued.

We have elected to wrap a function call around the increment of the outstanding counter to highlight the issue of atomicity between the main computation and the handlers. Handlers execute atomically with respect to other handlers, but may interupt computation. Thus, updates to shared variables at the computation level should be protected relative to handlers. Since the current version of *CMAM* uses polling, this concern does not arise. However, it is anticipated that interupts will be enabled in the future.

Figure 6 shows a weaker set of shared memory primitives based on the release consistency model. `write_iw` initates the write without waiting for previous writes to complete. `release` can be used directly to ensure that all previous writes have completed, or `write_i` will implicitly perform a release.

The two phases of `read_i` have been split apart in `prefetch_i` and `accept_i` so that global reads can be overlapped with computation. Notice that the previous shared memory primitives can be mixed freely with the weaker forms, except that no reads can occur between a prefetch and an accept. This restriction could be eliminated by providing separate buffers or, better yet, providing the buffer address as an argument to the prefetch. This idea is carried forward more thoroughly in the distributed memory model discussed in the next section.

```
/* Implementation of shared memory operations on CMAM */

volatile int sm_flag = 0;
volatile int read_buf;
inline void release() {while (sm_flag != 0) CMAM_poll();}
inline void outstanding() {sm_flag++; };

/* Read INT from a global address: <proc,addr> */
void read_i_resp_handler(int data)
{
        read_buf = data;
        sm_flag--;
}

void read_i_handler(int ret_node, int *local_addr)
{
        CMAM_reply_4(ret_node, read_i_resp_handler, *local_addr);
}

inline int read_i(int node, int *addr)
{
        release();
        outstanding();
        CMAM_4(node, read_i_handler, CMAM_self_address, addr);
        release();
        return(int_buf);
}

/* Write INT to a global address: <proc,addr> */

void write_ack_handler() { sm_flag--; }

void write_i_handler(int ret_node, int *local_addr, int data1)
{
        *local_addr = data1;
        CMAM_reply_4(ret_node, write_ack_handler);
}

inline void write_i(int node, int *addr, int data)
{
        release();
        sm_flag++;
        CMAM_4(node, write_i_handler, CMAM_self_address, addr, data);
}
```

Figure 5: *Shared Memory Implmentation.* File `shared-mem.c`.

```
/* Weaker forms of write and read overlap of communication and computation */

inline void write_iw(int node, int *addr, int data)
{
        sm_flag++;
        CMAM_4(node, write_i_handler, CMAM_self_address, addr, data);
}

inline int prefetch_i(int node, int *addr)
{
        release();
        sm_flag++;
        CMAM_4(node, read_i_handler, CMAM_self_address, addr);
}

inline int accept_i(int node, int *addr)
{
        release();
        return(int_buf);
}
```

Figure 6: *Weak shared memory operations.* File `weak-mem.c`.

4 Distributed Memory

This section describes a novel communication paradigm that falls somewhere between shared memory and message passing. It retains the global address space of shared memory, but exposes the split-phase nature of accessing data across a network. In addition, bulk transfer operations are provided. The idea basically is to allow the program to initate tranfers and test explicitly for their completion. Get generalizes read and put generalizes write. For integers, these are implemented as in Figure 7. This is part of the shared memory utility provided with the *CMAM* release as `CMAM_shmem.h`.

Note that the put handler is the same as the get response handler. The blocked version of get illustrates the use of communication segments to support bulk communication.

Figure 8 shows how the put operation can be used in supporting all-to-all personalized communication. The program is a generalization of a matrix transpose operation and forms of a key step in many fast parallel algorithms. The idea is that the layout of an array is specified by a mapping function from indexes to $<$ processor, offset $>$ pairs and the inverse. An array is provided under one layout and a new array is required under a different layout. In this case, mapping function A specifies a cyclic mapping and mapping function B specifies a blocked mapping. Each processor simply puts each datum where it goes. It is complete when all the data it requires has been put to it by others. There is no need to retaining consistency of write ordering within the xpose, no need for acknowledgments, and no barrier.

An alternative way of writing this routine would be to open a specific communication segment on all processors using `CMAM_open_this_segment`. The puts are then replaced by `CMAM_xfer` operations which can pack more data in a network packet, since the offset is relative to the base of the communication segment and no handler address field is needed. By organizing the remap to operate on quad words the full network bandwidth can be utilized.

```
/* Get data from remote processor and increment local flag */
inline void CMAM_get_i(int node, int *remote_addr, int *local_addr, volatile int *flag)
{
        extern void CMAM_get_i_handler();
        CMAM_4(node, CMAM_get_i_handler,
                CMNA_self_address, remote_addr, local_addr, flag);
}

/* Get block of data from remote processor and increment local flag */
inline void CMAM_get_N(int node, void *addr, int count, void *local_addr,volatile int *flag)
{
        extern void CMAM_get_N_handler();
        extern int CMAM_get_N_end();
        CMAM_4(node, CMAM_get_N_handler, CMNA_self_address, addr, count,
                CMAM_open_segment(local_addr, count, CMAM_get_N_end, (void *)flag));
}

/* Put data to remote processor and increment flag there */
inline void CMAM_put_i(int node, int *addr, int data, volatile int *flag)
{
        extern void CMAM_put_i_handler();
        CMAM_4(node, CMAM_put_i_handler, addr, data, flag);
}

void CMAM_put_i_handler(int *addr, int data, int *flag)
{
        addr[0] = data; (*flag)++;
}

void CMAM_get_i_handler(int ret_node, int *addr, int *ret_addr, int *flag)
{
        CMAM_reply_4(ret_node, CMAM_put_i_handler, ret_addr, *addr, flag);
}

void CMAM_get_N_handler(int node, void *addr, int count, int seg_addr)
{
        CMAM_reply_xfer(node, seg_addr, addr, count);
}
int CMAM_get_N_end(volatile int *flag, void *local_addr)
{
        (*flag)++; return 0;
}
```

Figure 7: *Distributed Memory Implmentation.* File `dist-mem.c`.

```
/* xpose.c - Generalized Transpose
 *
 * The layout of an array is specified by a pair of decode functions which
 * map the logical index (i) to a processor (p) and a processor local
 * offset (o).  The reverse mapping from (p,o) to i is given by an encode
 * function.
 *
 * Given two arrays specifed by arbitrary layout functions, move one array
 * into the other.
 *
 * Algorithm: scan through local elements, starting from a random offset,
 * and put them where they belong. Stop when have received a full set of puts.
 */

#include <cmam/cmam.h>

#define SIZE (1024)                        /* hard coded array size for now */

/* A Layout: wrap indexes around the processors. */
static double A[SIZE];
#define decode_a_p(i) ((int) (i % num_procs))
#define decode_a_o(i) ((int) (i / num_procs))
#define encode_a(p,o) ((int) (o * num_procs + p))

/* B Layout: block indexes onto processors. */
static double B[SIZE];
#define decode_b_p(i) ((int) (i / SIZE))
#define decode_b_o(i) ((int) (i % SIZE))
#define encode_b(p,o) ((int) (p * SIZE + o))

void CMPE_xpose(void)
{
        int me, num_procs, a, a0, r;
        static int count = 0;

        me = CMNA_self_address;
        num_procs = CMNA_partition_size;
        srandom(me);

        a = a0 = random()%SIZE;
        do {
                int vi = encode_a(me,a);
                CMAM_put_d(decode_b_p(vi), B+decode_b_o(vi), A[a], &count);
                a++; if(a == SIZE) a = 0;
        } while(a ≠ a0);
        CMAM_wait(&count, SIZE);
}
```

Figure 8: *Node program for the generalized array transpose example.*

5 Send and Receive

For those wedded to send&receive, here's a first-cut implementation: it only supports blocking send&receive (i.e. SEND blocks until the corresponding RECEIVE is executed, and vice-versa) and it only supports receiving from a specific node (i.e. cannot RECEIVE from "any node").

This implementation uses a standard three-way handshake: the sender sends a request to the receiver, which replies by acknowledgement when ready, and finally the sender transfers the data. The protocol state is maintained by the receiver in a per-processor table. `CMAM_send` sends a request Active Message to the receiver and, depending on whether the corresponding `CMAM_receive` has been executed, the request handler either enters the request into the state table or directly replies with the acknowledgement. In the meantime, `CMAM_send` waits for the acknowledgement before sending the data itself. Similarly, `CMAM_receive` checks the state table and, if the corresponding request has been received, sends the acknowledgement.

```
/* State of pending sends and receives */
#define MAX_PROC 1024                                     /* max num of processors */
static int state[MAX_PROC];                                                /* state */
#define IDLE -1                                                   /* processor idle */
#define SEND_PEND(state) ((state) >= 0)          /* he's waiting to send <state> bytes */
#define RECV_PEND -2                                     /* we're waiting for him to send */

/* State of current SEND */
static volatile int send_seg = -1;                                 /* segment to send to */
static volatile int send_count = 0;                            /* agreed message length */

/* State of current RECEIVE */
static volatile int recv_flag = 0;                              /* receive-complete flag */
static int recv_done(void *info, void *base) { recv_flag++; return 0; }
static int recv_from = 0;                                          /* source processor */
static int recv_count = 0;                                                   /* length */
static int recv_seg = 0;                                                    /* segment */

/* CMAM_send - SEND to remote node */
void CMAM_send(int node, void *buff, int byte_count)
{
        /* send request to remote node – wait for ack – send data */
        send_seg = -1;                                                  /* no segment yet */
        CMAM_4(node, send_handler, CMAM_self_address, count);                 /* send REQ */
        while(send_seg == -1) CMAM_poll();                                 /* wait to get ACK */
        CMAM_xfer(node, send_seg, buff, send_count);                         /* send data */
}
```

Figure 9: *Implementation of blocking send&receive (part 1).*

```
/* Handle a send request */
static void send_handler(int requesting_node, int send_count)
{
        if(RECV_PEND(state[requesting_node])) {                 /* is receiver ready? */
                recv_from = requesting_node;
                /* message length is MIN(send_count, recv_count) */
                if(send_count < recv_count) {
                        CMAM_shorten_segment(recv_seg, recv_count-send_count);
                        recv_count = send_count;
                }
                /* send ACK */
                CMAM_reply_4(requesting_node, send_set_seg, recv_seg, recv_count);
                state[requesting_node] = IDLE;                  /* ready for next */
        } else {
                state[requesting_node] = send_count;    /* .. not ready, record request */
        }
}

/* Handle send acknowledgement */
static void send_set_seg(int seg, int count)
{
        send_count = count; send_seg = seg;
}

/* CMAM_recv - RECEIVE from remote node */
void CMAM_recv(int node, void *buff, int count)
{
        /* allocate a segment */
        recv_count = count;
        recv_seg = CMAM_open_segment(buff, count, recv_done, 0);
        if(recv_seg == -1) CMPN_panic("no segment left");
        if(SEND_PEND(state[node])) {
                /* sender is already there */
                recv_from = node;
                /* message length is MIN(send_count, recv_count) */
                if(state[node] < count) {
                        CMAM_shorten_segment(recv_seg, count-state[node]);
                        recv_count = state[node];
                }
                /* send ACK */
                CMAM_reply_4(node, send_set_seg, recv_seg, recv_count);
                state[node] = IDLE;                             /* ready for next */
        } else {
                state[node] = RECV_PEND;
        }
        /* wait for completion of receive */
        CMAM_wait(&recv_flag, 1);
}
```

Figure 10: *Implementation of blocking send&receive (part 2).*

6 Mixing CMAM and CMMD

While not recommended, it is possible to mix both *CMAM* and CMMD in the same program with some care:

- CMAM and CMMD node-node communication cannot occur a the same time: CMAM cannot deal with the arrival of a CMMD packet and vice-versa. It is possible to use either one in well separated phases of the program though.
- The CMMD operations using the CM-5 control network are compatible with CMAM except for `CMAM_barrier`. Enabling the barrier changes the "host participation" in control network operations and interferes with CMMDs use of the control network.

A Manual Pages