```
}
```

In this code segment, we are attempting to read **blkno** of RAM into **buffer**. First, the block is checked to see if it has been bound to a server. Then, the block is requested from either the original site, or the mirror-site if the original site returns FAILURE, in the **mirror_getBlks** function. Now the function waits until the flag **retval** is set, signifying completion of the am_get(). If an error occurs, the server from which the block is being fetched is tested. If the server has gone down, then we continue by requesting the block from the remaining server.

6.0 Summary and Conclusions

In this paper we have explained why implementing GAM upon TCP/IP sockets is non-trivial: the TCP/IP reliability and sequenced byte stream doesn't buy much! We have also listed problems with GAM encountered while implementing a prototype network RAM server. Specifically, we proposed two new functions: am_sleep and am_isalive to allow blocking until message arrival and detecting node failure, respectively.

The TCPAM implementation was certainly no more difficult than CMAM or HPAM, but the TCP/IP guarantees makes it deceptively easy. In fact, any message layer that GAM is built on needs to supply the five requirements including reliability, N to 1 queue, memory-memory transfers, and deadlock free communication,

5.2 Error Model

The current GAM specification does not include an error model. Thus, when an error is encountered TCPAM calls exit! Clearly this is not acceptable for an application that desires fault tolerance. We have implemented a very simple error model consisting of error propagation and a new function, **am_isalive**.

In the updated TCPAM, all errors are propagated to the callers in the form of a FAILURE return value and a global variable, am_errno, that reflects the global errno value at the time of failure. This provides the architecture for detecting that an error has occurred. In order to gain specific knowledge about an error, an addition is proposed to GAM in the form of an **am_isalive** function. This is a predicate that tests the status of a remote machine.

Function: am_isalive()

int am_isalive(vnn_t vnn)

The function checks the socket associated with the vnn, and returns 1 upon success or 0 on failure.

5.2.1 An example: Page mirroring in network RAM

In order to increase the reliability in a network ram server, the server must at least know when a failure occurs in order to invoke the backup method(e.g. page mirroring). In our prototype implementation of network ram, we use the defined error model in the following code section:

```
int mirror_read(int blkno, char *buffer)
{
  volatile int retval;
  PageLoc *pageLocp;
  int server;

  while (numBlks > 0) {
    retval = 0;

    /*Check to see if the block segment is bound anywhere in NRAM*/
    if (mirror_lookup(blkno)==FAILURE)
      return FAILURE;

  try_again:
    if ((server=mirror_getBlks(blkno,buffer,&retval))==INVALID_VNN)
      return FAILURE;

    while (retval == 0)
      if(am_poll() == (FAILURE))
        if (!am_isalive(server))
          goto try_again;

    blkno++;
    numBlks--;
}
  return(SUCCESS);
```

1. Both nodes send the store packet header and block when writing the data
2. Both nodes poll to service the network, and receive the header
3. Now both nodes wait for the rest of the store data. Deadlock.

Solution 2: Modify polling function

A simple solution would be to enforce polling of only the reply network when sending a request. However, this yields the same conclusion as not polling the network at all! Thus, we need a poll that doesn't block. **am_poll** was modified to return control if only partial requests were received; the header packets are stored in a temporary location and the data packets are partially stored to the destination addresses. These two solutions have given us (we hope) deadlock free TCPAM.

5.0 GAM 1.1

In addition to the problems cited in TCPAM, we have found that GAM 1.1 lacks the functionality needed in an application such as network ram. For example, GAM lacks:

- A way to sleep until message arrival
- An error model

5.1 Sleeping until message arrival

Most active message applications (including Split-C) can afford to poll() messages at regular intervals to drain the network and effectively eliminate deadlock. Another class of applications, servers, require constant attention to the network to decrease response time to clients' requests. Polling the network constantly is too expensive to be practical, and regular polling may not be attentive enough. Thus an application would like to sleep until message arrival. We have implemented a new function, am_sleep(), which blocks on a socket select call until an incoming message arrives at one of the specified file descriptors.

Function: am_sleep()

void am_sleep(void)

Blocks until a message arrives on the network.

5.1.1 An example: multithreaded network RAM server

Two threads comprise the network RAM server. One is blocked waiting for requests from the kernel for page-in/page-out operations. The other, the network thread, simply is in an infinite loop that am_sleep()'s then am_poll()'s. Thus, the server can immediately respond to any page-out/ page-in request from a client without the cost of busy-waiting.

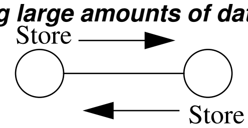
4.4 Deadlock

It would seem that such a TCPAM implementation would suffice given the above constructs for virtualizing an N to 1 queue and polling. More directly, the following can be asked:

Question: Given that TCP/IP provides reliable and ordered data streams, why isn't a TCPAM implementation trivial?

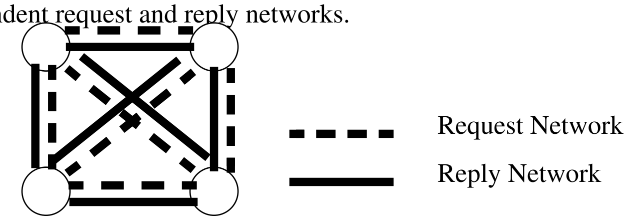
The answer to this question lies in a basic principle of networks: the network must be continuously drained in order to prevent deadlock. Specific to active messages is one of the four key issues proposed by von Eicken: send failure. In order to prevent deadlock, the network must continuously be drained even in the presence of send failure. Thus, when sending requests or replies, both handlers and user code must be prepared to service the network. In the implementation proposed above, there are several situations when the network cannot be drained in the presence of send failure. We will draw one example that summarizes the two changes deemed necessary for a deadlock-free active message implementation.

4.4.1 Example: *Two nodes storing large amounts of data to each other*



In the implementation of TCPAM described thus far, neither node can service the network if blocked on a socket write call. If the network were polled, an incoming request message could be serviced which could generate a reply. The reply message would mix with the outgoing data stream, which we actually observed in our network ram server! Thus, if the nodes were to block when writing data (because of buffer limitations) then deadlock would be achieved; each node would be waiting for the other to service the network.

Solution 1: Create independent request and reply networks.



This is a well-known solution and exists in many hardware platforms, including the CM-5. This effectively solves the above problem, and allows both nodes to now poll the network without fear of mixing data on the request network (since all incoming messages will only generate replies). However, we can still create deadlock given the following sequence of events:

4.0 Building GAM

The challenge of any GAM implementation is to identify the semantic differences between the lower-level substrate and active messages. Given the GAM semantics above, a general (but not complete) list of functionality requirements is: (This is a subset of the complaints listed in the CAL whitepaper.)

- N to 1 queue
- Ability to name and store (get) directly to (from) remote virtual addresses
- No events associated with messages
- Deadlock free communication
- Reliable communication

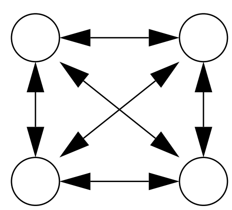
TCP/IP sockets provide *only* the last requirement! Thus, the other four must be built from the socket primitives.

4.1 TCP/IP Implementation:TCPAM

The four requirements of GAM that are missing from TCP/IP sockets are grouped into three categories: N to 1 queue, ‘active’ messages, and deadlock free communication.

4.2 Virtualizing an N to 1 Queue

Since a socket is limited to pair-wise communication, each node must have a separate connection path to all other nodes. This creates a complete graph with nodes as vertices and socket connections as edges, as shown below:



In order to create a virtual N to 1 queue, all sockets must be scanned when am_poll executes.

4.3 ‘Active’ Messages

TCP/IP sockets do not provide a means for signalling message arrival, so the message ‘queue’ must be polled as in other versions of GAM. Each message is broken up into a header and an optional data packet (for store requests and get replies). The polling function analyzes the header information and calls the appropriate handler. Since there is no notion of a memory-memory transfer, the store data follows the header and is incorporated into the user’s address space via the socket read function.

am_store stores **num** bytes from **local_va** to **remote_va** at the **destination_node**. When the store completes, **store_handler** is invoked with an optional argument on the remote node.

- am_get(source_node,remote_va,local_va,num,get_handler,arg0)

  am_get reads **num** bytes from **remote_va** at **source_node** and stores the data at **local_va**. When the transfer is complete, **get_handler** is invoked on the local node with an optional argument.

3.0 Overview of TCP/IP Sockets

TCP/IP sockets provide a reliable and sequenced communication path. Each socket is a connection that allows two hosts to communication via a full-duplex byte stream. Several operations can be performed against the byte stream, including read and write. Other utility functions exist to create a connection and check if a stream is empty. A summary of some socket functions follows:

- socket(family, type, protocol)

  Allocate a file descriptor in the open file table for a socket connection. The socket **family** specifies the address family such as internet(ip addresses) or unix domain (path names). Socket **type** can be either datagram or stream, while the **protocol** is usually limited by the socket type.

- bind(socket, address, sizeof(address))

  Bind an **address** in the address family to **socket**

- connect(socket, server_address, sizeof(server_address))

  Attempt to connect the named **socket** to another node specified in the **address** structure.

- accept(socket, client_address,sizeof(client_address))

  Accept a connection via **socket**, and fill in the **client_address** structure. A new file descriptor specifying the accepted socket is returned.

- read(socket, local_va, num)

  This procedure will attempt to read **num** bytes from **socket**, and store the result at **local_va**. The number of bytes actually read is returned.

- write(socket,local_va,num)

  Write() attempts to store **num** bytes starting at **local_va** to **socket**. The number of bytes actually written is returned.

- select(numbits, readlist,...,...., time-out)

  Select() checks a list of file descriptors associated with **read-list** and return the number of sockets that have messages waiting. **numbits** specifies the maximum number of file descriptors to check. **Time-out** points to a structure that indicates the amount of time to block on a select. A null pointer indicates infinite time.

White Paper on Building TCP/IP Active Messages

Lok Tin Liu, Alan Mainwaring, Chad Yoshikawa

In this paper, we briefly describe the implementation of tcpam, focusing on the modifications necessary to remove deadlock and add a simple error model. An example application built on tcpam, network ram, is given.

1.0 Introduction

Although TCP/IP provides a reliable and sequenced byte-stream, there are many difficulties associated with building Generic Active Messages (GAM) on this substrate. Many of these problems arise from the differences in communication models provided by sockets and GAM. In this paper, we analyze these problems and describe an implementation of active messages built on sockets (TCPAM). In addition, we offer some proposed changes to the GAM interface as motivated by our work on network ram.

2.0 Overview of GAM

Generic Active Messages (GAM) is a low-level communications architecture based upon request and reply ‘active’ messages; the messages themselves not only transfer data but also control. The GAM library consists of five primitive operations: request, reply, store, get and poll. Below is a summary of the semantics of each primitive:

- am_request (destintation_node,request_handler, arg0,..., arg4)

  This procedure requests that the **destination_node** invoke the function pointed to by **handler** with the specified arguments. Handlers can be called with 0 to 4 arguments.
- am_reply (requesting_node, reply_handler, arg0,..., arg4)

  This procedure can only be invoked from within the **request_handler** noted above. A message is sent which instructs the **requesting_node** to invoke the **reply_handler** with the arguments listed.
- am_store(destination_node, local_va, remote_va,num,store_handler,arg0)