# HPAM: An Active Message layer for a Network of HP Workstations

**Richard P. Martin**
**University of California at Berkeley**
**rmartin@CS.Berkeley.EDU**
**December 19th, 1995**

## Abstract

This document describes an Active Message layer we have constructed on a Network of Workstations. Active Messages is a thin, highly optimized communication layer targeted at the library or compiler writer. A primary goal of an Active Message layer is to deliver the minimum latency and peak bandwidth of the network hardware to user programs. Previous work on Active Messages demonstrated an order of magnitude performance improvement over vendor supplied send and receive libraries for Massively Parallel Processors.

The HPAM layer currently runs on a network of 4 HP 9000/735 workstations with Medusa FDDI interface cards. HPAM achieves a round trip time for a 20 byte payload of 29 µsec, an order of magnitude better than traditional software, and a maximum bandwidth of 12 MB/s.

## 1. Introduction

Recent technological trends have driven Massively Parallel Processor (MPP) hardware towards systems that are essentially a collection of workstation class nodes connected by a high performance network. Advances in Local Area Network (LAN) technologies can provide a Network of Workstations (NOW) with switched, high bandwidth MPP like interconnects, blurring the line between an MPP and a NOW. Traditional communications software for a LAN, *e.g.* TCP/IP, has lagged behind MPP software in terms of the fraction of raw hardware performance it can deliver to programs. The low performance stems from the assumptions made in traditional LAN software. These include invoking the operation system on every message, driver support for complex gather/scatter operations, and protocols which model communication only in point-to-point terms rather than in an all-to-all framework. Studies show that for TCP, only a small fraction of the communication time is spent accessing the actual hardware. Protocol processing consumes substantial time, but still it is only a small fraction of the total cost. The costs of all operations of the communications software, including context switching, buffer and timer management, scheduling, and data copying must be reduced to improve performance [2] [6]. The HP Active Message layer (HPAM) is a software layer which delivers close to the hardware performance to user level programs on a NOW without sacrificing essential services of the communication layer. The layer takes positions which differ from typical LAN communications software. These are:

- Direct user access to the hardware.
- An all-to-all, request-reply model of communication.
- Careful management of the network state to keep the overhead low.

This document describes HPAM. Section 2 provides background information. Section 3 describes the requirements of the HPAM layer. Section 4 introduces the Active Message abstraction. Section 5 details the HPAM implementation and Medusa interface. Section 6 presents a simple state transition diagram which implements the protocol used in HPAM. Section 7 presents performance measurements and a brief comparison to other software layers. The conclusion discusses future directions.

## 2. Background

Previous implementations of Active Messages on the Ncube and CM-5 assumed an MPP environment. HPAM, however, was constructed as a prototype Active Message layer for a NOW environment. HPAM addressed two of the primary differences between an MPP environment and NOW. First, while in an MPP environment the network is viewed as a reliable, HPAM views the network as ‘nearly reliable’. That is, the network error rates are very low, but not low enough that the computation can be aborted in the event of an error. Second, the HPAM assumes that network data may arrive for a process which is not running, unlike the Ncube version of active messages which space-shared the machine. The CM-5 the network is strictly gang scheduled among users, thus the operating system guarantees a message cannot arrive for an inactive user process.

HPAM was built using the Medusa network interface card. It is an Fiber Distributed Data Interface (FDDI) card with a large amount of on-board memory. Section 5.1 describes the Medusa in greater detail. The Medusa card presented an interesting interface compared to the Ncube and

CM-5. The Ncube network is accessible via Direct Memory Access (DMA) only. In the opposite extreme, the CM-5 network interface can only be accessed by Programmed IO (PIO). The Medusa card lies somewhere between these two extremes. The card has 1 megabyte of on-board VRAM divided into 128 discrete 8k blocks. Unlike the simple FIFO interface of the CM-5, the Medusa VRAM is addressable as a RAM. In the Ncube, queues were built in software out of the RAM used by the DMA engines. The Medusa, however, has a set of hardware FIFO pointers to the VRAM, thus queues do not have to be constructed in software.

A key question when first constructing HPAM was if the large amount of on board storage could be mitigate the added overheads of reliability and protection. Section 5 describes how the HPAM implementation leveraged the VRAM to this purpose. Section 7.2 presents these costs in greater detail.

## 3. Requirements

This section describes the capabilities that HPAM provides to user level programs. HPAM is designed as a set of mechanisms upon which higher level abstractions can be built *i.e.*, it is expected to be used by compilers or library writers. We view the following as essential to meet this goal:

**Composability.** The implementation of higher level abstractions on top of HPAM must be simple and straightforward. For example, in order to facilitate construction of a shared memory abstraction HPAM provides safe (deadlock and livelock free) request-reply protocols. Other important abstractions include message passing and byte streams.

**Reliability.** A reliable interface must be presented to the higher level. HPAM handles all bookkeeping, time-outs and retransmissions. A failure to deliver a message after a reasonable amount of effort is viewed as a catastrophic error. Presenting a reliable interface greatly improves composability as well.

**Protection**. The HPAM implementation must protect two non-cooperating programs from interference. A parallel program consists of co-operating processes which may be physically distributed on separate processors. HPAM provides trusted communication within a parallel program and protection between parallel programs.

**Efficiency**. A simple and consistent communication cost model must be presented to a compiler or library writer. For example, HPAM abstracts a short message as register-to-network operation, and a long message as a memory-to-network operation. A compiler can determine when it is appropriate to use the short or long version, given the cost of the register move vs. the memory copy.

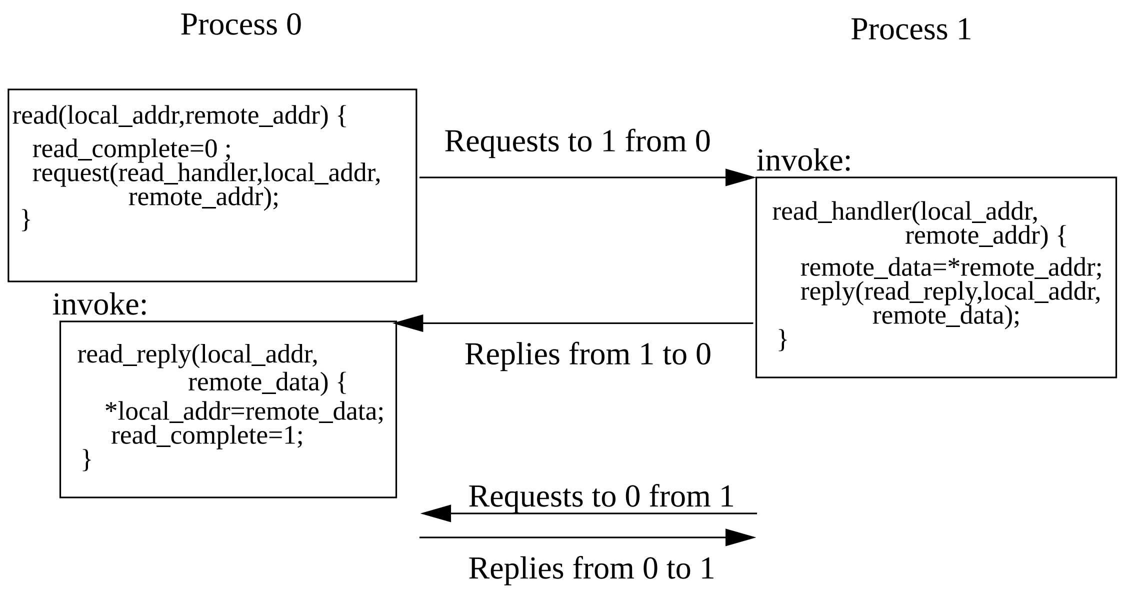
We did not view ordering as an essential property because it is not required for all higher level communication abstractions. For example, for a shared memory abstraction with explicit completion events (Split-C) [3], ordering is not as critical a property to support as reliability [10]. Ordering can be implemented cheaply on top of HPAM since the programmer is freed from reliability concerns.

## 4. Abstraction: Active Messages with Request-Reply

Active messages present a simple mechanism to the programmer: each message contains the address of a user-level handler (code segment) which is executed on message arrival. Because handlers run at the priority of the network, thus preventing the servicing of successive messages, they must run quickly and to completion [11]. The role of the handler is to get the message out of the network, either by integrating it into the computation or sending a reply. Programming with active messages is similar to programming interrupt handlers in the operating system. The programmer must reason about asynchronous functions invoked within a main body of code.

In addition to the basic active message mechanism, HPAM enforces a request-reply model of communication. When invoked, a handler is typed by the system as either a request handler or reply handler. A request handler may only use the network to issue a reply to the sending process. Reply handlers cannot use the network. Request handlers which do anything other than reply, as well as reply handlers which attempt to use the network, generate an error.

Traditional models, *e.g.* TCP, assume there are two logical one-way streams of information between two communicating entities. The request-reply model, however, assumes there are two, two-way streams. Figure 1 illustrates this using a remote *read* operation built upon HPAM. In the example, process 0 initiates the read to fetch a value stored in the address space of process 1. First, the 'main thread' of control clears a completion flag, then launches a request. On process 1, the read_handler is called when the message arrives from the network. The read_handler function accesses the value then sends a reply back to process 0. When the reply arrives the read_reply handler is invoked and stores the value into the appropriate location. The read_reply handler also sets a completion flag so the 'main thread' can detect the completion of the read. In a like manner,



```
read(local_addr,remote_addr) {
  read_complete=0 ;
  request(read_handler,local_addr,
             remote_addr);
}
```

```
read_handler(local_addr,
               remote_addr) {
   remote_data=*remote_addr;
   reply(read_reply,local_addr,
            remote_data);
}
```

```
read_reply(local_addr,
              remote_data) {
   *local_addr=remote_data;
   read_complete=1;
}
```

**FIGURE 1. Logical information flow in request-reply model**

process 1 can issue remote reads to process 0.While the restrictions on handlers may seem confining, in practice they have not been. The restrictions were motivated by the need to provide an efficient interface on top of an unreliable network.

HPAM presents separate abstractions for small and large messages. A small message lives in the registers defined by the calling convention used by compilers for the instruction set. The send operation for a small message is a register-to-network operation. Conversely, receiving becomes a network-to-register operation. The *HPAM_4* and *HPAM_reply_4* calls pass four words from the local call frame to a remote call frame because the PA-RISC calling convention dedicates four registers for passing arguments. See Appendix A for the code which implements the *HPAM_4* function.

The *HPAM_xfer* and *HPAM_reply_xfer* functions support large messages, *i.e.,* bulk transfers. The model they present is a remote memory copy with a notification on the receiver that the copy has completed. HPAM_xfer copies bytes from the local process's memory, starting at the local base address, to the remote processes's memory starting at the remote address. Once the copy is completed, the handler is invoked on the remote node with a pointer to the data, the length, and a user supplied argument. Likewise, request handlers can use HPAM_reply_xfer to transfer bulk data back to the requesting process. HPAM does not provide segmentation and re-assembly because many transfers will fit into a single FDDI packet, which is up to 4500 bytes. Higher layers can compose segmentation and re-assembly easily since HPAM guarantees packet delivery.

# 5. Implementation

This section explains the current HPAM implementation. It begins with a quick overview of the Medusa card, then describes the all-to-all request-reply protocol used, and how it provides safety. The section concludes by explaining how the finite resource model exposed by the HPAM protocol can be used to provide protection. The Appendix shows the C code for two of the most commonly used HPAM functions, the *HPAM_4* and *HPAM_poll*.

## 5.1 Medusa overview

The Medusa card was designed for high bandwidth TCP connections [1]. The card sits on the graphics bus (SGC bus) in place of the frame buffer, and has one megabyte of on-board VRAM. The VRAM is divided into fixed sized (8 Kbyte) blocks, each of which may contain one packet. Four memory mapped FIFOs control access to the card (see Figure 2). An entry in a FIFO encodes a <block number/length> pair for one packet in VRAM. Stores are pipelined on the SGC bus, and can be issued every 2 cycles. On a 99 MHz HP 9000/735 with a 33 MHz Medusa the latency of a load instruction from the VRAM is 42 cycles. The bandwidth available from main memory into the card is 38 MB/s. Using straightforward programmed I/O, the bandwidth from the Medusa VRAM into main memory is only 10 MB/s. With the assistance of a special block-move unit, the processor can move data at a rate of 18 MB/s from VRAM to main memory.

To send a packet is simple. The processor constructs the FDDI header and payload in a block on the VRAM. The processor then stores the encoded <block number/length> pair to the TX_READY FIFO, causing the card to start transmitting the packet into the network. Once the

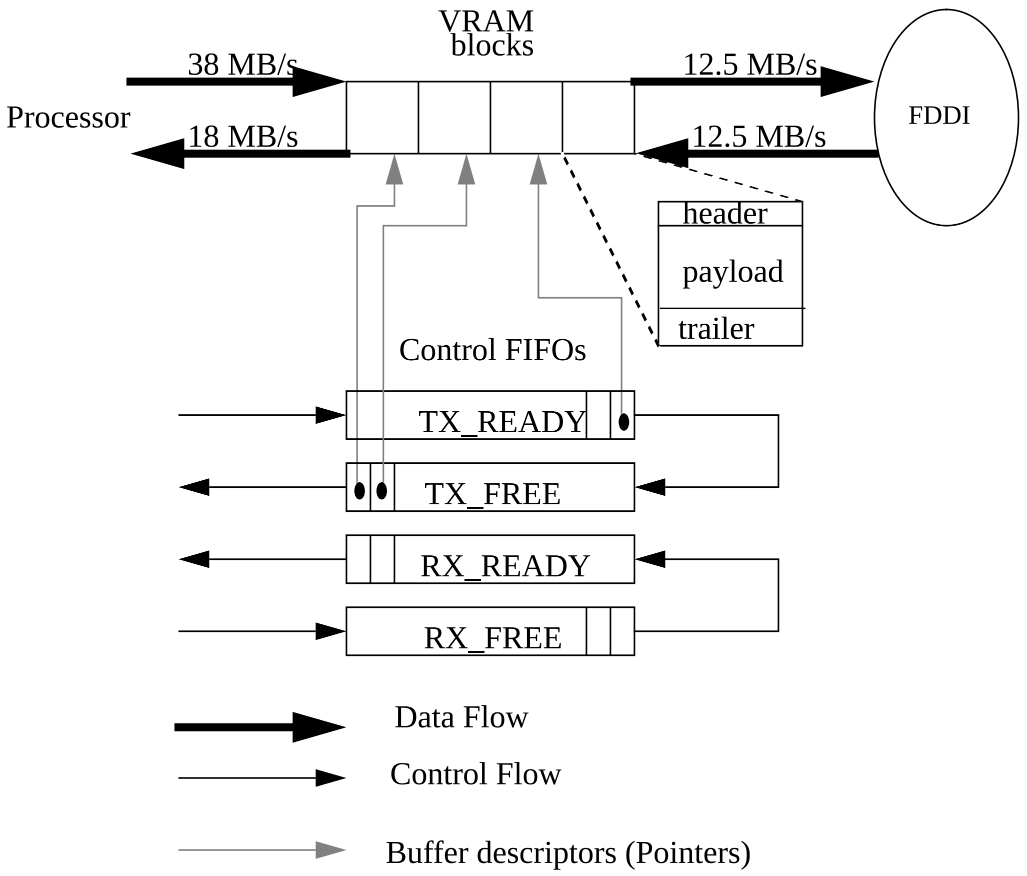


**FIGURE 2. Medusa Card**

packet has been transmitted, the card places the block number on the TX_FREE FIFO. The VRAM is not altered by packet transmission.

When receiving a packet, the card pops the next entry off the RX_FREE FIFO and writes the FDDI header, payload and trailer into the corresponding block of VRAM. If RX_FREE is empty, the card drops the packet. Once the entire packet arrives, the Medusa places the descriptor on the RX_READY FIFO. To receive a packet, the processor loads the next <block number/length> pair from the RX_READY FIFO, and then accesses the corresponding VRAM block. Assuring the atomicity of message transmission and reception is straightforward because a single load or store instruction sends or receives a packet. The FIFOs control ownership of the blocks. Blocks in the TX_READY and RX_FREE FIFOs are logically owned by the card; the processor logically owns blocks in the TX_FREE and RX_READY FIFOs. However, nothing in the card hardware checks for the consistency of the FIFOs. The card can interrupt the processor when the RX_READY FIFO goes from empty to non-empty.

## 5.2 HPAM implementation

This sub-section describes the current HPAM implementation on the Medusa. Although the Medusa uses FDDI as the physical layer, the ideas presented in this work are independent of any particular physical layer. Many of the ideas in HPAM are applicable to communications software as long as the physical network exhibits the properties described below. First, the assumptions underlying the implementation and its novel properties are described. Next, a simple all-to-all request-reply protocol and an augmented higher bandwidth protocol are described, followed by a discussion of how the protocol is safe.

The HPAM layer is optimized under the following two key assumptions which differ from traditional LAN software:

- The probability of packet loss is very small, but not completely negligible.
- The network hardware will not duplicate packets or deliver packets to the wrong destination.
- The network characteristics (latency, bandwidth, round trip time) are roughly constant throughout the duration of the communication.

We believe these assumptions make sense in a high performance LAN environment where the communicating entities and network fall under the same administrative control. For example, the current implementation is optimized for case where the processes are constituents of a parallel program and are running at the same time. The common case is thus that the arriving message is for the currently running process, so no context switch is needed on message arrival.

The novel properties of HPAM are:

- The protocol assumes an all-to-all communications model, as opposed to a point-to-point model.
- The request-reply model allows for a low overhead implementation of active messages using simple tables. HPAM does not use pointer based data structures, instead it trades memory for computation.
- All state is explicit in the protocol, and kept at the endpoints. This forms the basis for the protection mechanism.
- A process owns the network device while it is running. HPAM maps the Medusa into the process's address space to avoid the overhead of a kernel trap. A scheduler external to the layer is responsible for protecting the network state of a process to allow multiple processes to share the card.

### 5.2.1 Low overhead: single request-reply protocol

This section describes an all-to-all, single request-reply protocol. The first HPAM prototype used this protocol, and it serves to illustrate the core ideas. The basic idea of the protocol is simple. In an all-to-all model, a communicating entity (process) communicates with **P**-1 other processes. Each process reserves **P**-1 request buffers (VRAM blocks), one for every other communication partner. A process also reserves **P**-1 reply buffers, one to store the reply for each request it may receive from another process. All request and replies are saved in buffers corresponding to the destination, and exist for the duration of the program. Both request and reply buffers are for sending, *i.e.,* out-bound messages. In addition, at least 2(**P**-1) buffers are reserved for in-bound messages. A simple linear data structure, the *descriptor table*, describes the state of each request and reply buffer. This state includes status, such as whether the buffer is in-use, the sequence number, and time-out information. In-bound buffers form a pool; they do not have descriptor table entries. The following example shown in Figure 3 walks though a request-reply operation, such as the remote read described in Figure 1.

1. Before launching a request to process *i*, HPAM acquires the request buffer reserved for process *i* (step 1.a). In the read example this would be the request buffer reserved for process 1. The request buffer is free if the last outstanding request received a reply. If the request buffer is free, HPAM constructs a packet (complete with sequence number, etc.) containing the request in the Medusa buffer and stores the pointer/length pair into the TX_READY FIFO to send the packet (step 1.b). Finally, HPAM marks the buffer as in-use and timestamps the request by copying the value of the interval timer register into the descriptor table entry (step 1.c). A cheap timer mechanism[1] is essential to maintain low overhead, as noted in [2]. Returning to the remote read example, the HPAM_4 call will store the argument registers of the call frame into the Medusa buffer then send and mark the buffer.

   If the request buffer is marked as in-use, the process enters a time-out and re-transmission loop, stalling on the request until the reply corresponding to the previous request that used this buffer is received. While stalled, the processor continues to service the network (both requests and replies), but will not return from the request. Time-outs and re-transmissions use an exponential back-off scheme. If the remote process does not respond within 30 seconds, HPAM gives up and signals an error.

2. The in-bound buffer pool (the RX_READY FIFO) holds all unprocessed messages. When a message arrives, the HPAM layer must first determine the type of the message (request or reply) and the sender in order to match the sequence number. All that is required to perform the matching operation (steps 2.a and b) is an index by the sender into the descriptor table followed by a compare. A significant number of other checks (detailed in section 5) must pass as well. If the match was successful, HPAM invokes the request handler (step 2.c), then returns the buffer to the in-bound buffer pool by storing the block number in the RX_FREE FIFO. If the handler responds with a reply, the reply packet is stored in the reply buffer reserved for that remote process, overwriting the previous reply (step 2.e). In the read example, if the sequence numbers matched for the remote read request HPAM would next load the arguments to the read_handler call from the Medusa buffer into registers and branch to the read_handler function.

If a request handler does not reply, HPAM generates an empty acknowledgment automatically. Request handlers that do anything but reply to the sender cause an error and the operation returns a failure code.

1. On the HP 735, the PA- RISC interval timer is a 32 bit register updated every cycle [5]. Using the interval timer greatly reduces the cost of timer management; the system call *gettimeofday* costs 18 µsec, while the interval timer costs 1 cycle to read.

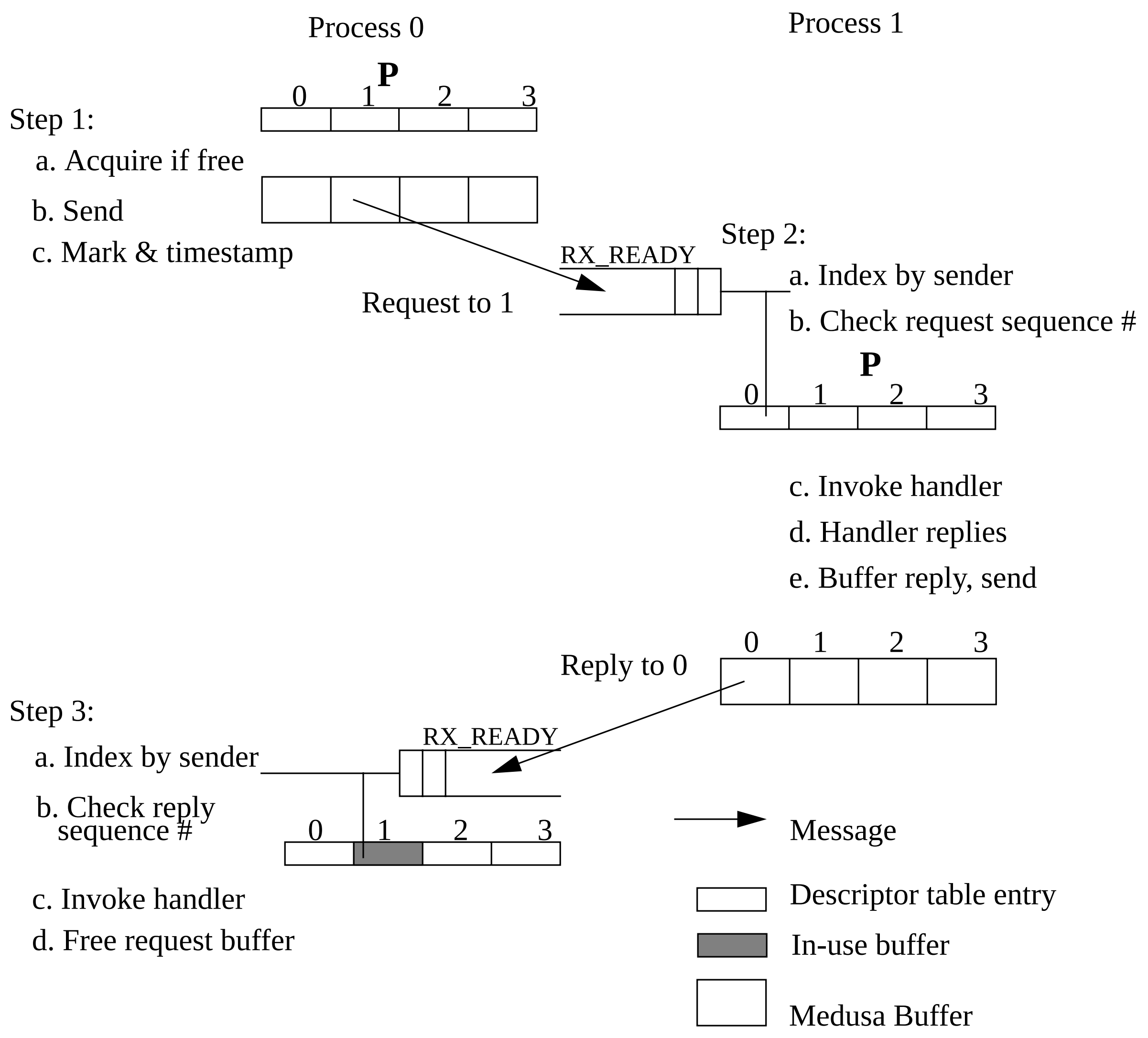


**FIGURE 3. Connection state maintained by single request-reply protocol**

3. When the reply arrives, HPAM must again match the sequence number based on the sender and type of the message (step 3.a and b). If the sequence numbers match, the reply handler is invoked (step 3.c). After the reply handler returns, the corresponding request buffer is marked as free, invalidating the timestamp (step 3.d). Reply handlers which attempt to use the network generate an error.

The protocol is very similar to an alternating bit protocol; the sequence numbers only need to range from 0-1 for the protocol to work. The following paragraphs describe how the protocol protects against loss and duplicate packets.

A free request buffer means that both the previous request and reply were successful, so a requesting process can re-use the request buffer. A valid new request means the previous reply got through, so a process can re-use the reply buffer.

An invalid request means that either the corresponding reply was lost or the requestor timed-out and sent a duplicate. The receiver, however, cannot distinguish a dropped reply from a duplicate request. Therefore when receiving an invalid request the correct action is to re-send the previously stored reply. Suppose in the example read, the packet containing the read reply was dropped by the network. Process 0 would time-out waiting for the reply, and re-send the request. Process 1 would have processed the first request and have the reply stored in the reply buffer. Upon receiving the duplicate request, HPAM would 'reflect back' the previously reply instead of invoking the handler. Handlers cannot be invoked twice, because an active message may not be idempotent. Consider a fetch-and-add operation rather than a simple read. If the requestor timed-out and sent two requests, HPAM would send two replies, causing the requestor to see a duplicate reply, which it drops.

The protocol will never lose packets due to buffer overflow. A most **P**-1 requests can be outstanding to a given process at a time. By reserving 2(**P**-1) buffers in the in-bound pool, all possible requests and replies can be held pending in the Medusa card. [1] In the worst case, all processes may make a request to the same process, consuming **P**-1 in-bound buffers. That process in turn may make **P**-1 requests as well, requiring an additional **P**-1 in-bound buffers for replies.

HPAM uses polling instead of interrupts because of the high cost of an interrupt. See Appendix B for the code which implements the *HPAM_poll* function. The cost for an interrupt and the kernel's first level interrupt service routine is 10 μsec. All HPAM functions automatically poll, but the user must be careful in compute only loops, since the process is ignoring the network.

Since the combination of the descriptor table and Medusa buffers hold all the network state, HPAM implements reliability by checking the time-out value of a single request descriptor for each poll. If the descriptor has timed out, HPAM doubles the length of the timer and resends the message. A continuous circular walk though the requests in the descriptor table is enough to ensure all packets get re-transmitted in case of loss. Replies never time-out since they are only re-transmitted in responses to requests.

### 5.2.2 Full bandwidth: multiple request-reply protocol.

Although the simple request-reply protocol has very low overhead, it cannot realize the full network bandwidth between pairs of processes because each pair can have only one outstanding request. To obtain the full network bandwidth, HPAM replicates the single request-reply protocol. The number of replications needed to realize the full bandwidth of the network between pairs of nodes is the network depth, **D**. For the Medusa, **D**=4. Most a high performance LANs have a small hardware network depth.

1. Due to a bug in the MAC chip of the Medusa, HPAM may actually drop a packet due to buffer overflow. For every packet sent on the FDDI ring, the MAC chip generates a status packet in the RX_READY FIFO of all stations on the ring. Thus, a station's RX_READY FIFO can fill up with status packets even if the station is not communicating.

The new protocol is an all-to-all **D**-way request-reply protocol. The protocol is very similar to the single request-reply protocol. The descriptor table is now 2-dimensional, with indices by processor and instance number. The basic match must include the instance number of the protocol (see Figure 4) as well as the processor and sequence number. Because HPAM replicates the single

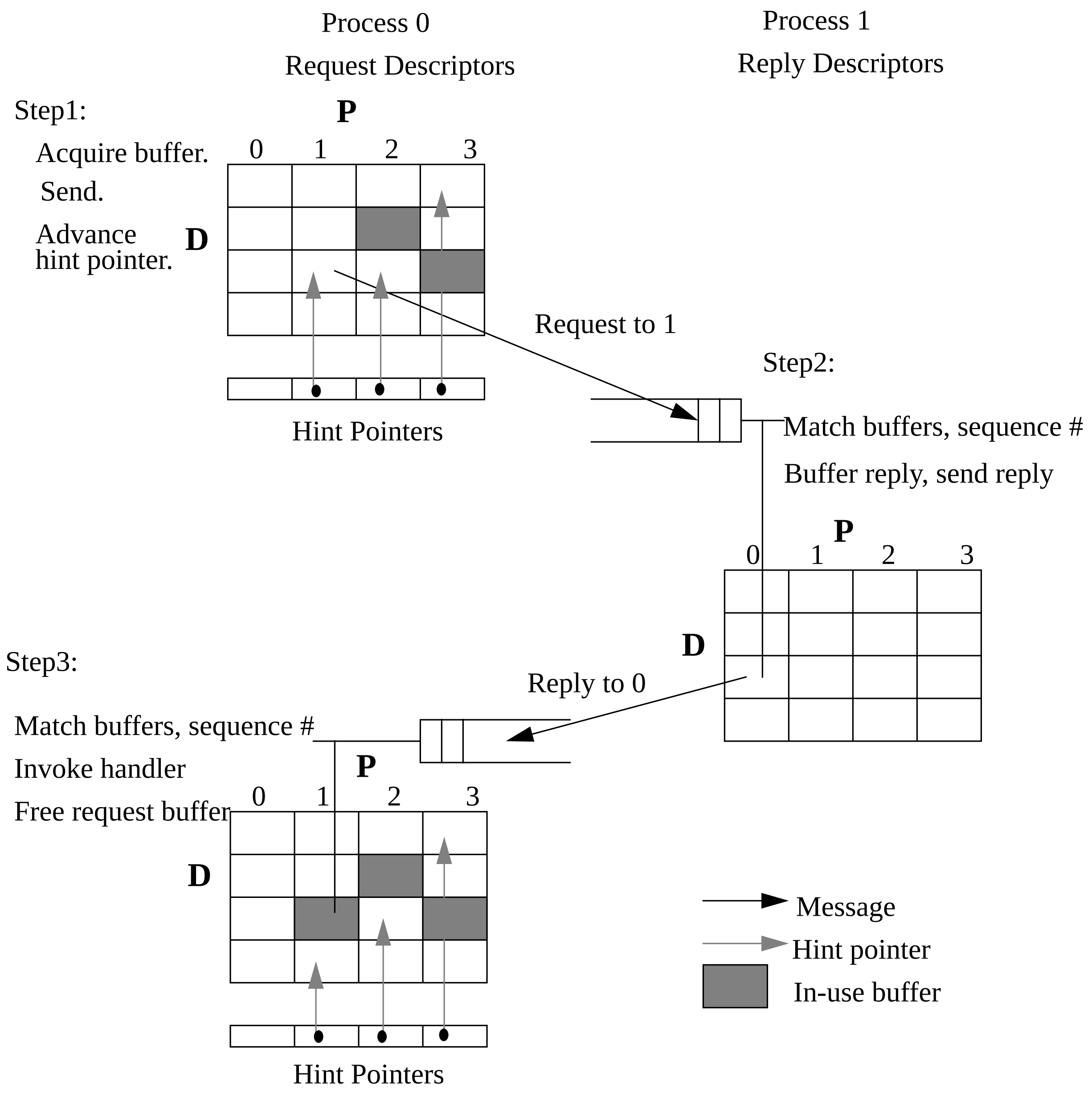


**FIGURE 4. Connection state maintained by the all-to-all D-way protocol**

request-reply protocol, it does not preserve ordering.

The new operations for a remote read are:

1. First, acquire a free request buffer. Any buffer for the desired destination will do, so HPAM maintains a ‘hint pointer’. If the buffer pointed to by the hint is not free, HPAM searches the descriptor table, starting at the top in linear order, for a free buffer. If no free buffer is found, the request stalls and enters a time-out loop waiting for the hint to point to a free buffer.
2. The tuple that defines a valid match is now a triple: <processor, instance, sequence number>. Thus, the match is logically a two-dimensional lookup. Each of the **D** instances of basic protocol have a distinct sequence number space. Thus HPAM is not preserving ordering of the requests. Imagine the read request was dropped. HPAM will later re-transmit the request packet (containing the read_handler), but that will not prevent HPAM from invoking sucessive request handlers. However, HPAM still guarantees all handlers are invoked exactly once.
3. When a reply frees a request, the hint pointer is changed to point to the newly freed buffer. If the processes was stalled on a request, changing the hint will allow the stalled request to immediately use the newly freed request buffer.

The above protocol provides a simple form of flow control; at most **D** requests to a given processor can be outstanding at any given time. The overhead needed to support flow control in HPAM is very low, as shown in section 5. Any form of flow control must have a mechanism to free resources based on arriving messages. Because HPAM enforces a request-reply model, the match needed to implement the flow control is kept extremely simple. Instead of complicated pointer structures, HPAM maintains a simple two dimensional table (see Figure 4). The two dimensional nature of the protocol means every process must reserve *O*(2***P*****D**) space for out-bound communication. In addition, *O*(2***P*****D**) Medusa buffers must be reserved as in-bound buffers to prevent packet loss due to buffer overflow.

HPAM obtains reliability in the same way as the single-request reply protocol, except that the walk through the request descriptors is 2-dimensional.

In normal operation, if replies come in from the network in order and no packets are lost, each hint pointer cycles sequentially through the a column of the descriptor table. The hint pointer always points to a free buffer unless the processor has used up its allocated bandwidth. In that case, the processor may unnecessarily search the descriptor table, however, the process cannot send anyway. Complicated schemes to buffer the message and try again later would add a significant amount of overhead without improving bandwidth.

Many protocol implementations always put out-bound data on a time-out list, only to remove it later when the acknowledgment arrives without the packet ever timing out. HPAM expects packet loss to be the infrequent case. By not maintaining an explicit time-out list, HPAM makes the frequent case faster, at the expense of the infrequent case when a packet is dropped.

### 5.2.3 Request-reply: solving deadlock.

Any request-reply protocol must guard against possible deadlock or livelock arising from the two-phased nature of the operation. This problem is described well for networks which guarantee delivery in [8]. For networks which may drop packets, the problem manifests itself as follows.

Suppose a single pool of buffers holds all in-bound and out-bound data, and all data is buffered at the sender until an acknowledgment is received. Between two processes, both buffer pools may become full, each process requiring a reply from the other to free some buffers. However, neither process can send a reply because the buffer pool is full. Thus, the two processes are deadlocked. If the underlying layer attempts to re-transmit the requests, the requests must be dropped since the buffer pools are full; the processes become livelocked. HPAM is safe because no process can attempt to use the network unless resources, both request and reply buffers, are available for the entire path of a communication. Since reply handlers cannot use the network, a path consists of at most two hops.

## 5.3 Protection

HPAM recognizes a parallel program as a collection of mutually trusting processes and provides protects parallel programs from each other. A key is associated with a given parallel program. All out-bound messages from a process of the program are stamped with the key. For in-bound messages, the key must match the process's key before a message can be accepted by the process.

HPAM relies on a scheduling daemon, external to the layer, to ensure that only one process may use the Medusa at a time. The process given access to the Medusa is the 'active process', and is allowed to run. The scheduling daemon stops all other processes which need to use the Medusa. The daemon swaps network state into and out of the card when switching the active process. To save the network state, the daemon walks the descriptor tables, copying out all active state out of the Medusa buffers and into a per-process save area. Restoring the state is the reverse, copying the old state back into the Medusa. The scheduler must copy the active state because HPAM maps the card into the process's address space to avoid the overhead of a kernel trap.

The scheduler does not guarantee that all arriving messages are for the active process, although this is the expected case. The HPAM library and the scheduler negotiate two queues per process: an input queue the process uses to accept messages from the scheduler, and an output queue the process uses to send messages to the scheduler. The scheduler associates with each process the key which tags a message as valid for a given process. Before accepting a message, the HPAM library checks the key. If the key in the message mismatches the key of the active process, the message is copied into the output queue. After the daemon suspends the active process, the daemon copies all messages in the output queue of the suspended process into the correct processes' input queues. A process just swapped in by the daemon first checks its input queue for messages before checking the Medusa.

In the current HPAM implementation, there is a small critical section just after the process pops the RX_READY FIFO when it may be suspended by the scheduling daemon. In this critical section, the HPAM layer knows the VRAM block number of the received message, but the daemon does not. The daemon is unable to restore the state of the block when changing the network process. One solution would be to copy the entire contents of the VRAM if the process was suspended in the critical section. The current implementation sets a received failed flag, which must be checked after each pop from the RX_READY FIFO. If the flag is set, HPAM will drop the packet. In future implementations the daemon will manipulate the process' state, much like a debugger does, to obtain the needed information.

Although the current protection mechanisms are adequate in our research environment, they assume a malicious user does not modify the HPAM code or data structures. Mechanisms exist in the PA-RISC hardware for providing both protected code and data without invoking the operating system [5].

## 6. Logical buffer state transitions

The purpose of this section is to illustrate how a hardware implementation of the state transitions could accelerate HPAM. The buffer management in the HPAM protocol can be abstracted into a simple state transition diagram. The state transitions described in this section were chosen to illustrate the set of logical transitions; HPAM takes some shortcuts.

First, the section describes how the all-to-all single request-reply protocol described in section 5.2.1 maps to simple state transitions. Next, the section describes how data structures amenable to a hardware implementation could implement the state transition diagram.

## 6.1 Logical states and transitions

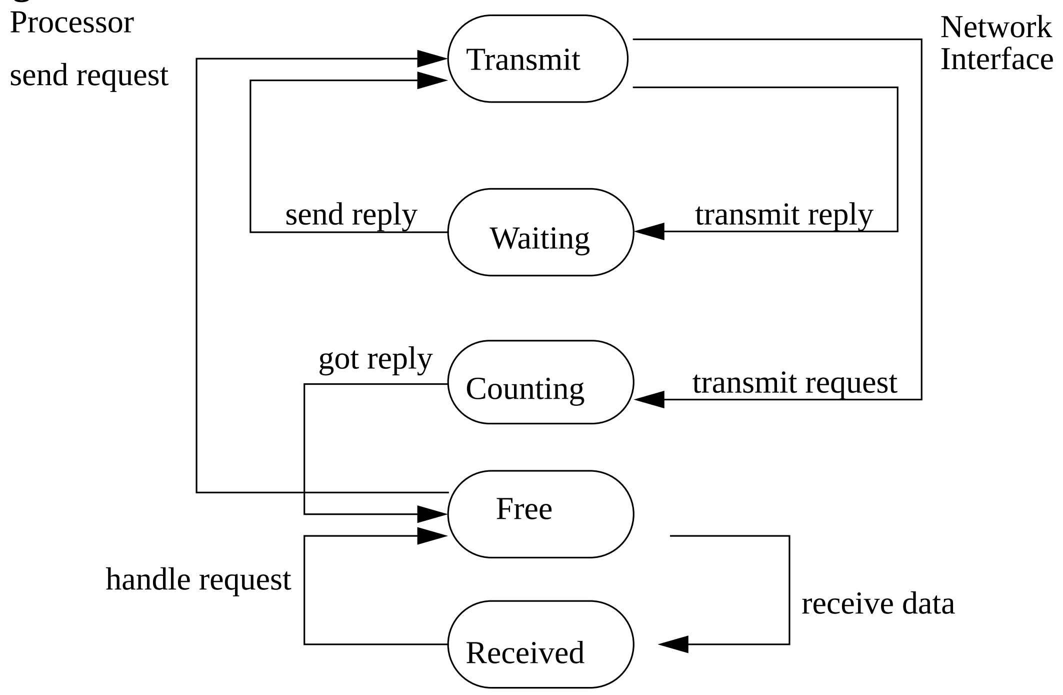


**FIGURE 5. State transition of VRAM buffers**

Figure 5 shows the five states of a VRAM buffer. These are: transmit, waiting, counting. free and received. Table 1 shows the meaning of each state.

| State | Status of the VRAM buffer |
|---|---|
| Free | Available for use |
| Transmit | Queued for transmission over network |
| Counting | A timer is counting down for this buffer |
| Waiting | Buffer is not available for re-use. |
| Received | Contains data queued from the network |

**TABLE 1. States used in HPAM.**

Following the actions for the single request-reply protocol described in section 5.2.1, when the processor issues a request, the VRAM buffer allocated for the request moves from the free state to the transmit state. Unlike the transitions shown in Figure 5, in HPAM the processor updates the state of the VRAM buffer (entries in the descriptor table) when the buffer moves into the counting or waiting state. In addition, the processor does not wait for the buffer to transition out of the transmit state before updating the descriptor table. This optimization is possible because the matching reply or request cannot be issued until the buffer has left the transmit state.

When a request arrives at a node, the Medusa moves a buffer from the free state to the received state. After the processor executes the handler, the buffer can be returned to the free state. The execution of the handler causes a reply or empty ack to be sent to the requesting node (steps 2.d and 2.e in Figure 3). When the processor issues a reply, the buffer moves from the free state to the transmit state.

In HPAM system, this corresponds to the processor marking the buffer as in use and storing the VRAM block number in the output FIFO (steps 1a and 1b in section 5.2.1). The processor also keeps track of buffer's timer. the From the Medusa's viewpoint, the processor has moved the buffer to the transmit state.

## 6.2 Hardware data structures used in transitions

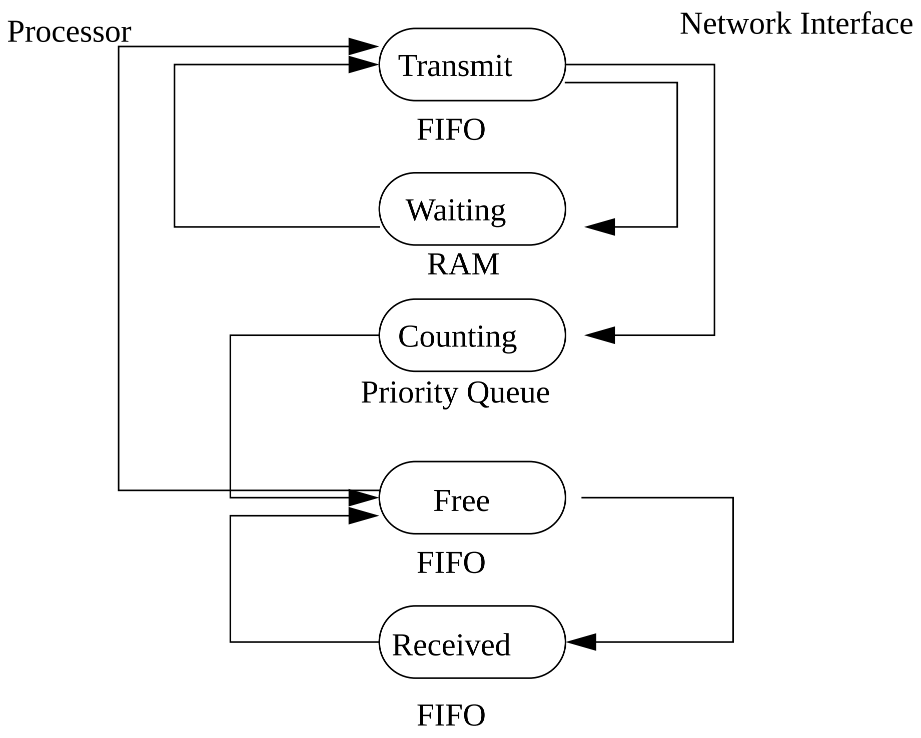


**FIGURE 6. Data Structures needed to implement state transitions**

Figure 6 shows a few simple data structures which could be used to implement the state transitions described in Figure 5. All of these data structures are amenable to a hardware implementation. Hardware which realized the model presented above would accelerate the most common HPAM code paths. Instead of managing the storage to keeping track of state transitions, an HPAM software layer built using such hardware would only need to command the hardware as to which buffers needed moving. The hardware would keep track of the state of each buffer.

All buffers are addressable as RAM by the processor. This is to allow arbitrary incoming requests free replies. The hardware data structures shown above need only store pointers to buffers. Like the Medusa card, the Transmit, Free and Received states can be implemented as hardware FIFOs of pointers to buffers. Buffers in the counting state can be organized in a priority queue. The queue is organized with the head to be the buffer with the smallest time-out value. The only operations the processor can perform against the queue are insert a value with it's priority equal to the time-out and delete. A systolic priority queue can be implemented such that insertions and deletions take only $O(1)$ time.[9].

## 7. Performance

This section documents the performance of HPAM, and compares it to TCP on the same hardware. First, the section begins with some quick definitions then presents a performance comparison between HPAM and two TCP implementations. Finally, a detailed measurements of the cost to send and receive a packet are presented.

Following LogP [4] terminology, *overhead* is defined as the processor cycles spent preparing to send or receive a message. *Latency* is defined as the time from when the processes signals the network interface that the message is ready to the time the remote network interface is prepared to hand the message to the processor.

For all the measurements, two 99 MHz HP 735 workstations were set up back to back forming a two node ring. A two station ring was used to be consistent with other measurements, and to reduce the network latency to a minimum. No programs were running other than normal system daemons during the tests. The stations were connected into the public Ethernet.

We measured the latency of the raw FDDI hardware as a fixed cost of 8.39 µsec plus 0.08 µsec/byte. The methodology used was to measure the round trip time (RTT) of increasing size packets. The software used was the bare-minimum needed for the test, not HPAM. One process sent a packet and started a timer. The card buffer was pre-loaded with the FDDI header. On the remote end, another process sat in a tight polling loop, sending a pre-loaded buffer to the first station as soon as it popped the packet from the RX_READY FIFO. For a given packet size, the experiment was repeated until a 95% confidence interval was achieved. Using a least squares fit, the time for a zero length packet represents the fixed overhead part, and the slope is the cost-per-byte. A zero payload FDDI packet consists of at least 22 bytes, so a lower bound on the latency an application can hope to see is 10.15 µsec

### 7.1 Comparison to TCP

This section shows a comparison between HPAM and two implementations of TCP on the same hardware. The two versions of TCP are the ‘normal’ version, and the single copy version. The single copy stacks are detailed in [1]. The single copy TCP/IP stacks differ from the normal protocol stacks in that they only make one copy of the data, and they take advantage of the Medusa’s special IP checksum unit.

Two experiments were performed, the first measures the round trip time, the second measures the bandwidth under streaming conditions. For both experiments, the TCP NODELAY option was set, and the socket buffer size was set to 56K bytes.

### 7.1.1 Round Trip Measurements

In this experiment, a packet is sent by one station, then echoed by another. The first station

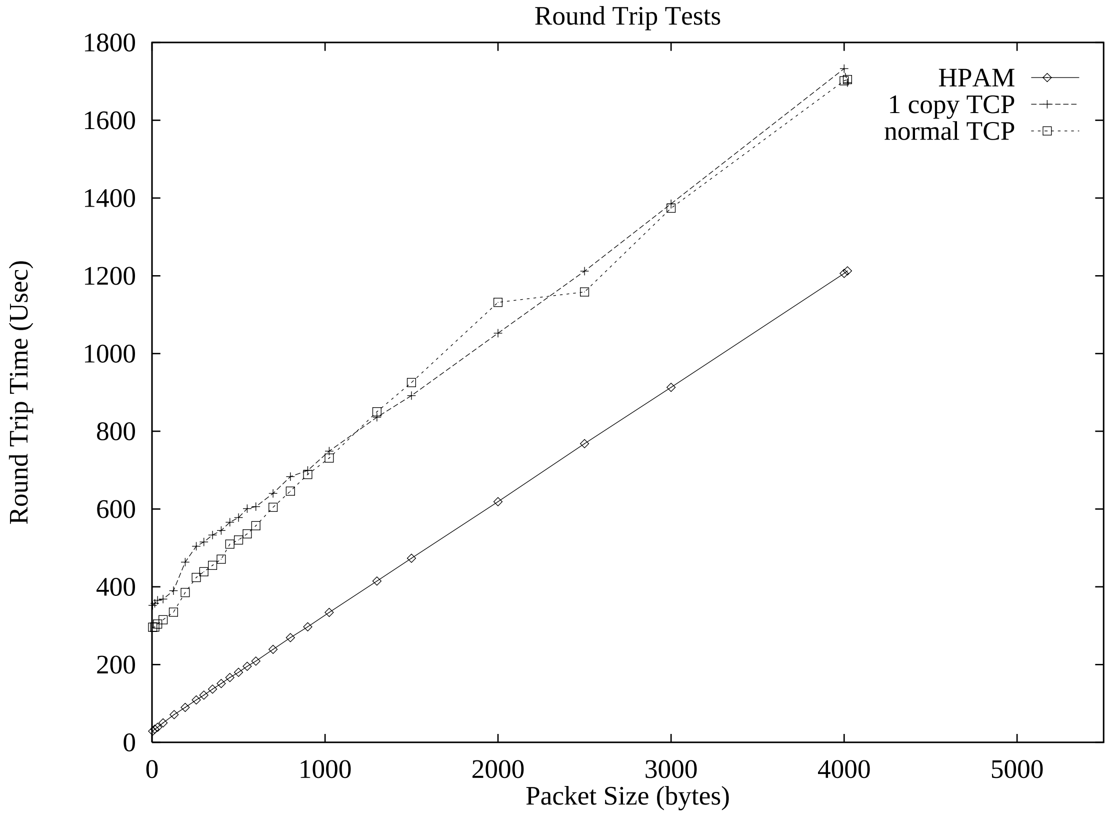


**FIGURE 7. Round Trip Time measurements**

records the round trip time. The experiment is repeated for the packet sizes up to 4K bytes. This experiment models a request-reply type operation. Figure 7 shows the results. A single copy stack provides little benefit over normal TCP. The measurements show for this implementation, the single copy stacks have a higher overhead than regular TCP for packets up to 1K bytes. Keeping the overhead low in a request-reply operation is critical to obtaining bandwidth because the overhead cannot be overlapped with message transmission.

### 7.1.2 Bandwidth Measurements

The second experiment measures bandwidth under ideal conditions. The first station sends 5000

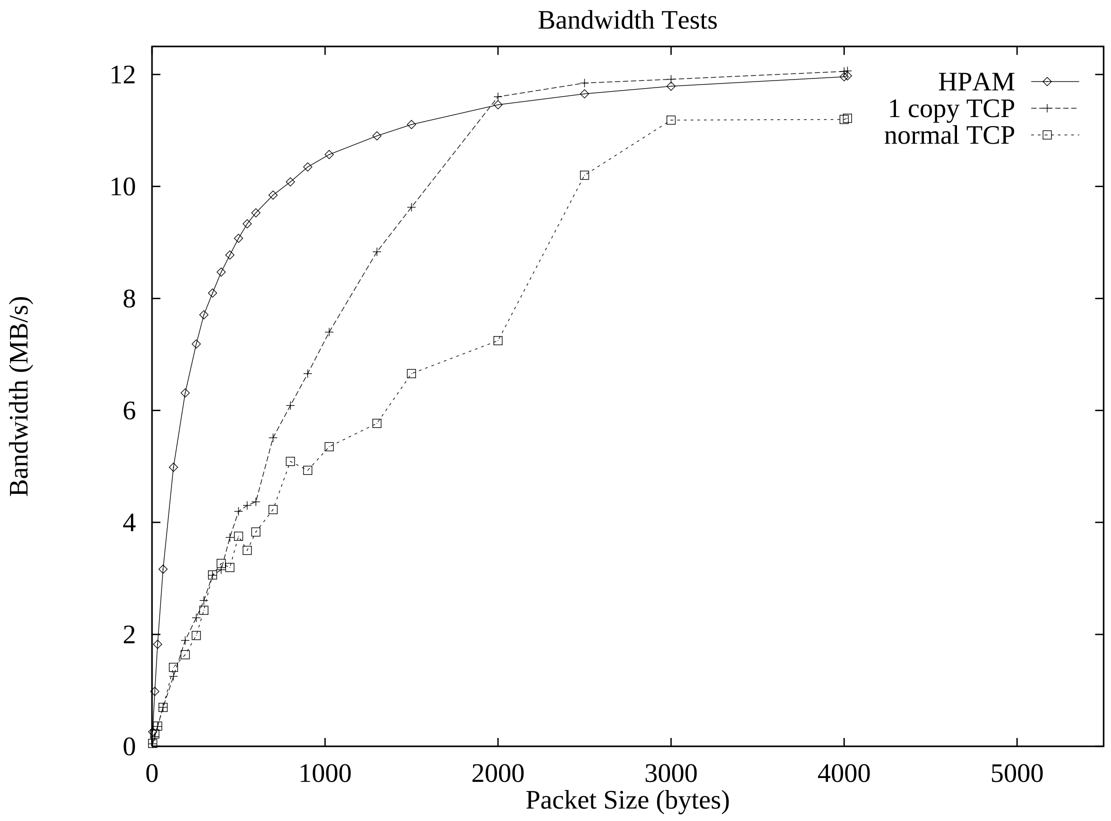


**FIGURE 8. Bandwidth measurements**

packets of a given size to the second station without waiting for an acknowledgment. The second station starts timing after receiving the first packet. After the last packet is received, the receiving station computes the bandwidth. This experiment gives the message layer the best possible chance for obtaining high bandwidth, because the layer can maximize the overlap between overhead and message transmission.

Since HPAM has very low overhead, it achieves the peak rate($R_\infty = 12$ MB/s) quickly, reaching the half-power point,($N_{\frac{1}{2}}$) at 176 bytes. For the TCP single copy stack, $N_{\frac{1}{2}} = 761$ bytes, and for the normal TCP stacks $N_{\frac{1}{2}} = 1352$ bytes. Notice from the measurements that the single copy stacks only provide a performance benefit in the regime between 800 and 3000 bytes. This is because with larger packet sizes, the rate of the normal TCP stacks becomes network limited, but for small packets is overhead limited.

## 7.2 Detailed Overhead Measurements

This section isolates the overhead of three features of HPAM: basic features, protection and flow control/reliability. Flow control and reliability count as a single feature because HPAM uses the same mechanisms to accomplish both. A ‘basic’ feature provides the minimal functionality of the HPAM_4 call without reliability or protection. The ‘FDDI’ feature represents a cost which is an artifact of FDDI. Table 2 shows the total costs of each feature.

| Feature | Cycle Count (send + receive) |
|---|---:|
| Basic (B) | 317 |
| Flow Control/Reliability (FC) | 245 |
| Protection | 219 |
| FDDI(FDDI) | 215 |

**TABLE 2. Cost By Feature**

Figure 9 shows the costs, in cycles, to send a message using the HPAM_4 call. The figure also shows the time an HPAM_poll call takes to extract the message and invoke the handler. The costs are broken down by step, and each step is assigned a feature.

On the sending side, the first step is to acquire a free buffer. In these tests, the first buffer examined was free. It is reasonable to ignore the cost to search for a free buffer because the hint pointer will always point to an available buffer unless a packet was dropped or the process has used all the request buffers. Next, HPAM marks the card buffer as in-use so it will be saved by the scheduler if the process is switch out. The packet is constructed in the card buffer and launched into the network during the ‘Store packet’ step. Most of the fields in the descriptor table can be updated after the packet is sent into the network; this is the ‘Update state’ step. The Medusa card returns a status packet for every packet sent. The status packet is simply discarded by HPAM. However, the cost must be charged to the send overhead because every send incurs at least one poll of a status packet. The “Service (poll)” step accounts for this cost. The cost to remove the status packet doubles the basic send cost.

HPAM must go though seven checks on the receiving side before invoking the handler. First HPAM sets a flag which signals the scheduler that the RX_READY FIFO is about to be accessed. After popping the FIFO, HPAM must check if it was just context switched back in, in which case the pop operation failed and HPAM discards the packet.

Next, the packet must be checked to see if it is a status packet from the ring, and the CRC is checked. The CRC check did not count as an FDDI artifact since many network interfaces only signal the processor of a bad CRC rather than discard the packet. The packet must pass the key check, then the type checks for request/reply and 4 word/xfer. The “dispatch” step is the cost to extract the handler address and four word payload followed by the branch to the handler. This cost is 140 cycles because of the long the latency of a reads from the VRAM.

After the handler returns to the HPAM layer, it checks if the handler replied, which is the “Sent ack?” cost. In this case, the handler replied so no ack is sent by the layer. Next, HPAM_poll checks a single request descriptor for time-out. In the current implementation, HPAM_poll

Send

| step | feature | cycle count |
|---|---|---|
| Acquire buffer | FC | 60 |
| Mark buffer | P | 20 |
| Store packet | B | 83 |
| Update state | FC | 18 |
| Service (poll) | FDDI | 185 |
| Total send overhead | | 366 |

Time

Service Network (poll)

| step | feature | cycle count |
|---|---|---|
| Lock Card | P | 13 |
| Pop FIFO & Unlock | P | 140 |
| Status packet? | FDDI | 30 |
| CRC ok? | B | 85 |
| Key match? | P | 46 |
| Request or reply? | FC | 56 |
| 4 word or xfer? | B | 9 |
| Dispatch | B | 140 |
| Sent ack? | FC | 11 |
| Timed out? | FC | 100 |
| Total receive overhead | | 630 |

**FIGURE 9. Cycle counts to send a message and service the network.**

returns to the top of the “Lock card” step after the “Sent ack?” step, checking for time-out only after the RX_READY FIFO has been completely emptied.

The current HPAM implementation is able to overlap some of the overhead with the network latency. In particular, the “Service (poll)”, “Sent ack”, and “Timed out” steps occur after a packet has been launched into the a network. The total cost of a round trip operation, in this case remote read of an integer, takes 29 µsec to complete. This compares favorably with the 20 µsec absolute minimum that is possible using the Medusa cards.

## 8. Conclusion

The Active Message model, combined with a strong typing of request and replies enables a low overhead implementation of HPAM. Although the network is unreliable, the request-reply protocol keeps the matching process simple and fast. Also, no complex buffering is required. The request-reply nature exposed to the compiler or library writer frees the HPAM layer from having to infer the pattern of communication. If the higher level is sending many replies, the HPAM layer can take advantage of the two way communication to provide reliability and flow control. Likewise, a user sending a one-way set of requests is signaling the layer that it must generate empty acknowledgments on behalf of the user. As the comparison to TCP shows, sending the empty ack packets costs almost nothing. HPAM achieves nearly the same peak bandwidth as the single copy TCP, even sending an empty ack for each data packet.

By making all the state explicit in the protocol, HPAM provides a simple protection model. Protecting communication becomes a matter of protecting state. Thus, HPAM can avoid costly kernel traps because an external scheduler is able to protect the network state.

HPAM is able to achieve high performance while maintaining key features by making different assumptions that traditional LAN software. HPAM however, does not replace TCP in the contexts of highly unreliable, variable delay networks. Networks which have these characteristics, typically WAN internetworks, require more sophisticated mechanisms. Mechanisms associated with TCP, such as slow start and estimating variance of the round trip time [7], are trying to “second guess” the network. In addition to sending data, TCP constructs and dynamically maintains a model of the network. We believe that with software enforced flow control and scheduling, a LAN network can be static enough so that a layer can assume the network characteristics to be fairly constant. A layer can exploit that knowledge to realize large performance benefits.

Future work on HPAM includes improvements in the negotiation between the scheduler and the HPAM layer. For example, the Unix *ptrace* call can be used to allow the scheduler access to the state of the network process. The PA-RISC gateway page mechanism should be used to provide protection to the HPAM code and data from the user level process without incurring the cost of a kernel trap. The layer should also be hand coded in assembly language to realize the lowest possible latency.

## Acknowledgments

The work is part of the Berkeley NOW project. The author wishes the thank Chris Dalton and Greg Buzzard at HP labs for answering many questions about the Medusa card, and Doug Ghormley for writing the scheduler. The author would like to thank David Culler, Kim Keeton and Remzi Arpaci for their comments on this document.

## References

1. D. Banks and M. Prudence, “A High Performance Network Architecture for a PA-RISC Workstation”, *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, Feb. 1993.

2. D. Clark, V. Jacobson, J. Romkey, H. Salwen, “An Analysis of TCP Processing Overhead”, *IEEE Communications Magazine*. Vol. 27, no. 6, June 1989.

3. D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken and K. Yelick, “Parallel Programming in Split-C”, *Supercomputing ‘93*, Portland OR. Nov. 1993. IEEE Computer Society Press.

4. D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, T. von Eiken. “LogP: Towards a Realistic Model of Parallel Computation”, In *Proceedings of the*

*Fourth ACM Symp. on Principles and Practice of Parallel Programming*, San Diego, CA May 1993.

5. Hewlett Packard Company, “PA-RISC 1.1 Architecture and Instruction Set Reference Manual”, Manual Part Number 09740-90039.

6. J. Kay and J. Pasquale, “A Performance Analysis of TCP/IP and UDP/IP Networking Software for the DECstation 5000”, Sequoia 2000 Technical Report #92/10, 1992.

7. S. Leffler, M. McKusick, M Karels, J. Quaterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Reading, MA: Addison-Wesley Publishing Company, 1989.

8. C. Leiserson, et al. “The Network Architechture of the Connection Machine CM-5”, *In Proceedings of the Symposium on Parallel Algorithms and Architectures*, 1993.

9. C. Leiserson, “Systolic Priority Queues,” *Conference of Very Large Scale Integration: Architecture, Design Fabrication*, California Institute of Technology, Jan. 1979

10. Luna, Steve. “Implementing an Efficient Portable Global Memory Layer on Distributed Memory Multiprocessors”, UCB Technical report UCB/CSD 94/#810, 1994

11. T. von Eicken, D. Culler, S. Goldstein, K. Schasuer, “Active Messages: A Mechanism for Integrated Communication and Computation”, In *Proceedings of the 19th International Symposium on Computer Architecture,* Gold Coast, Australia, May 1992

## Appendix A: Source Code for the *HPAM_4* function

```
int
HPAM_4(int dest, void (*fun)(), unsigned long arg0,unsigned long arg1,
          unsigned long arg2,unsigned long arg3) {
  register buf_desc *state_buf;        /* buffer descriptor */
  register unsigned long state_word;   /* tmp state word        */
  register unsigned long base;         /* points to head of packet */
  unsigned long id, free;              /* ID, buffer pair */
  unsigned long i;
  if (dest != HPAM_self_address)  {

    if ((hpam_req_state[dest][hpam_free[dest]].state)&TX_RD) {
      state_buf = &(hpam_req_state[dest][hpam_free[dest]]);
      free = hpam_free[dest];

    sendit:
      base = (unsigned long) state_buf->buf;

    /* lock buffer */
      state_buf->size = base+PKT_4_DATA+16;
      state_buf->state = (state_buf->state&(~TX_RD)) | TX_NR;
      state_word = state_buf->state;

    /* next ID number */
      id = (state_word+1)&ST_ID_MASK;
      out_word(base+PKT_CTL,ARGS4|REQ|(id<<ID_SHIFT)|free);
      out_word(base+PKT_FUN,fun);
      out_word(base+PKT_4_DATA,arg0);
      out_word(base+PKT_4_DATA+4,arg1);
      out_word(base+PKT_4_DATA+8,arg2);
      out_word(base+PKT_4_DATA+12,arg3);
      out_word(TX_READY_FIFO,base+PKT_4_DATA+16); /* send it */

      state_word = (state_word&(~ST_ID_MASK))|(id&ST_ID_MASK); /* id number */
      state_word = (state_word&(~RETRY_MASK)); /* retries = 0 */
      state_buf->state = state_word;

      hpam_free[dest]= (hpam_free[dest]+1)&(MAXDEPTH-1); /* inc & mod */
      GETITMR(state_buf->timestamp); /* set timestamp */
      HPAM_poll();
      return 0;
    }
    else {        /* We have to find a free buffer */
      for (i=0;i<MAXDEPTH;i++) {
      if ( (hpam_req_state[dest][i].state) & TX_RD) {
        state_buf = &(hpam_req_state[dest][i]);
        free=i;
        goto sendit;
      }
      }
      if ( stall(dest) != -1) {
      state_buf = &(hpam_req_state[dest][hpam_free[dest]]);
      free = hpam_free[dest];
      goto sendit;
      }
      else
      return -1;
    }
  }
  else{                 /* message is to self */
    hpam_reply_dest=dest;
    (*fun)(arg0, arg1, arg2, arg3);
    hpam_reply_dest=-1;
  }

  HPAM_poll();
  return 0;
} /* HPAM_4 */
```

## Appendix B: Source Code for the *HPAM_poll* function

```
void
HPAM_poll() {
  int i;
  unsigned long rx_buffer;           /* pointer from RX_READY     */
  unsigned long recv_start,recv_len; /* start address and length  */
  buf_desc *req_state;               /* pointer to request state buffer   */
  buf_desc *rep_state;               /* pointer to reply state buffer   */
  register unsigned long control;    /* control word in message   */
  unsigned long base;                /* frame to send ack         */
  unsigned long pid_s_d;             /* pid part of message       */
  unsigned long crc;                /* crc bit                    */
  int src;                           /* src node                  */
  int id,buf;                        /* id, buffer pair           */
  int len;                           /* data length of xfer       */
  unsigned long preamble;            /* preamble (in bytes) for xfer */
  unsigned long source,target;       /* pointers for xfer call    */

 start:
  if (*hpam_in_q) {
    rx_buffer = hpam_get_input();
    goto bypass;
  }

  *hpam_busy = MED_RX_BUSY;
  if (((rx_buffer = in_word(RX_READY_FIFO)) & 0x80000000) != 0) {
    *hpam_busy = rx_buffer;

    if (*hpam_recv_nack) {
      *hpam_recv_nack =0;
      goto release;
    }

  bypass:
    recv_len  = rx_buffer & PKT_LEN_MASK;
    recv_start = rx_buffer & ~(PKT_LEN_MASK);
    if (recv_len > 0x12) {/* is the message valid? */
      /* most status messages = 0xc, broadcast =0x18?*/
```

```
crc = in_word((recv_start+(recv_len-2))&~0x3);
pid_s_d = in_word(recv_start+PKT_DST_SRC);
control = in_word(recv_start+PKT_CTL);

/*is the MSB of the second to last byte==0 then we have a good CRC. */
if ((crc>>(((4-((recv_len-2)&0x3))<<3)-1) &0x1) ==0 ) {  /* good crc */

if ( ((pid_s_d&PKT_PID_MASK)>>PID_SHIFT) == hpam_pid) { /*pid check*/

src = (pid_s_d&PKT_SRC_MASK);
id = (control&ID_MASK)>>ID_SHIFT;
buf = control&BUF_MASK;

if ((control&NET_MASK) == REQ)  { /* a request */
  rep_state = &(hpam_rep_state[src][buf]);

  if (id ==((rep_state->state)&ST_ID_MASK)) {
    hpam_reply_dest = src;
    hpam_reply_buf = buf;

    if ( (control&ARGS_MASK) == ARGS4) { /* an HPAM_4 call */
      ( *( (void (*)()) in_word(recv_start+PKT_FUN)))
      (in_word(recv_start+PKT_4_DATA),
       in_word(recv_start+PKT_4_DATA+sizeof(unsigned long)),
       in_word(recv_start+PKT_4_DATA+2*sizeof(unsigned long)),
       in_word(recv_start+PKT_4_DATA+3*sizeof(unsigned long)));
    }
    else {  /* Xfer call  */
      target=in_word(recv_start+PKT_X_BASE);
      source=recv_start+PKT_X_DATA;
      preamble = (control&OFFSET_MASK) >> OFFSET_SHIFT;
      recv_len -= (PKT_X_DATA+FDDI_TRAILER+preamble);

      len=recv_len;

      /* med_vcopy crashes with unaligned data and len == 0 */
      if (recv_len > 0) {
      /* punt to med_vcopy if the data is unaliagned in the
```

```
             * pre or postamble, or if there is enough
             * to justify the overhead */

            if (preamble||(recv_len&0x3)||(target&0x3)||
                (recv_len>USE_VENOM_SIZE)){
              med_vcopy((char*) (source+preamble),(char*) target,recv_len);
            }
            else {      /* aligned and under the size */
              while (recv_len) {
                * ((unsigned long*)target)=in_word(source);
                source+=4; target+=4; recv_len-=4;
              }
              target-=len;
            }
            } /* end if len > 0 */
            (*( (void (*)()) in_word(recv_start+PKT_FUN)))
            ((void * ) in_word(recv_start+PKT_X_ARG),(void *) target, len);
          }

          /*next id (sequence) number */
          rep_state->state=(rep_state->state&(~ST_ID_MASK)) |
            ((id+1)&ST_ID_MASK);

          /* if nobody sent and ack, send one */
          if (hpam_reply_dest != -1) { /* send an empty ack */
            base = (unsigned long) rep_state->buf;
            out_word(base+PKT_CTL,ACK|REP|(id<<ID_SHIFT)|buf);
            out_word(TX_READY_FIFO,base+PKT_CTL+sizeof(long)); /* send it */
            rep_state->size = base+PKT_CTL+sizeof(long);
            hpam_reply_dest = -1;
          }
        }
        else {    /* wrong sequence number maybe a duplicate */
          out_word(TX_READY_FIFO,rep_state->size);
#ifdef DEBUG
          printf("Req wrong seq number, retransmit rep %d %d %d\n",src,id,buf);
#endif
        }
      } /* end if a request */
      else {            /* may be a reply, broadcast */

        if ( (control&NET_MASK)==REP )  { /* a reply */
          req_state =  &(hpam_req_state[src][buf]);

          /* this conditional is tricky. When we get the reply,we may */
          /* get two of the same id in a row. So have to */
          /* check the second condition */

          if (id ==((req_state->state)&ST_ID_MASK) &&
            (!(req_state->state&TX_RD)) ) {

            if(!(control&ACK_MASK)) {/* Not an empty ack */

            if (control&ARGS_MASK) { /* a HPAM_4 call */
              ( *( (void (*)()) in_word(recv_start+PKT_FUN)))
                (in_word(recv_start+PKT_4_DATA),
                 in_word(recv_start+PKT_4_DATA+sizeof(unsigned long)),
                 in_word(recv_start+PKT_4_DATA+2*sizeof(unsigned long)),
                 in_word(recv_start+PKT_4_DATA+3*sizeof(unsigned long)));
            }
            else {      /* Xfer call  */
              target=in_word(recv_start+PKT_X_BASE);
              source=recv_start+PKT_X_DATA;
              preamble = (control&OFFSET_MASK) >> OFFSET_SHIFT;
              recv_len -= (PKT_X_DATA+FDDI_TRAILER+preamble);

              /* punt to med_vcopy if the data is unaligned in the
               * head or tail portions, or if the target address is
               * unaligned, or if there is enough to justify the overhead
               */
              len=recv_len;

              if (recv_len >0 ) {
                if (preamble||(recv_len&0x3)||(target&0x3)||
                  (recv_len>USE_VENOM_SIZE)){
                         med_vcopy((char*)  (source+preamble),(char*)  tar-
get,recv_len);
                }
                else {/* aligned and under the size */

                  while (recv_len) {
                  * ((unsigned long*)target)=in_word(source);
                  source+=4; target+=4; recv_len-=4;
                  }
                  target -=len;
                }
              } /* if len > 0 */

              (*( (void (*)()) in_word(recv_start+PKT_FUN)))
                  ((void  *  )  in_word(recv_start+PKT_X_ARG),(void  *)  target,
len);
            }
            } /* not ack */

            /* free buffer */
            req_state->state = (req_state->state|TX_RD);
            hpam_free[src] = buf;

          } /* sequence # ok */
          else {  /* wrong seq number */
#ifdef DEBUG
            printf("Rep from %d wrong seq number, drop\n",src);
#endif
          }
        } /* end reply sequence */
        else {    /* a broadcast packet */
          if (control&ARGS_MASK) { /* a HPAM_4 call */
            ( *( (void (*)()) in_word(recv_start+PKT_FUN)))
            (in_word(recv_start+PKT_4_DATA),
             in_word(recv_start+PKT_4_DATA+sizeof(unsigned long)),
             in_word(recv_start+PKT_4_DATA+2*sizeof(unsigned long)),
             in_word(recv_start+PKT_4_DATA+3*sizeof(unsigned long)));
          }
          else {  /* bcast xfer call  */
            target=in_word(recv_start+PKT_X_BASE);
            source=recv_start+PKT_X_DATA;
            preamble = (control&OFFSET_MASK) >> OFFSET_SHIFT;
            recv_len -= (PKT_X_DATA+FDDI_TRAILER+preamble);

            /* punt to med_vcopy if the data is unaliagned in the
             * head or tail portions, or if the there is enough
             * to justify the overhead */
            len=recv_len;
            if (recv_len >0 ) {
            if (preamble||(recv_len&0x3)||(target&0x3)||
                (recv_len>USE_VENOM_SIZE)){
              med_vcopy((char*) (source+preamble),(char*) target,recv_len);
            }
            else {      /* aligned and under the size */

              while (recv_len) {
                *((unsigned long*)target)=in_word(source);
                source+=4; target+=4; recv_len-=4;
              }
              target -=len;
            }
            } /* if len > 0 */

            (*( (void (*)()) in_word(recv_start+PKT_FUN)))
            ((void * ) in_word(recv_start+PKT_X_ARG),(void *) target, len);
          } /* xfer */
        } /* end bcast sequence */
      }
      } /* end if pid check */
      else{
#ifdef DEBUG
      printf("HPAM_poll(): got packet for NPID %d size %d\n",pid_s_d,
             recv_len-FDDI_TRAILER);
#endif
      hpam_buffer(recv_start,recv_len);
      }
    }  /* end if crc is good */
     else {
       fprintf(stderr,"HPAM_poll(): received packet with a bad CRC\n");
     }
    } /* end if not a status packet */
    if (rx_buffer >= med_card_base)  /* is the buffer in I/O space? */
      out_word (RX_FREE_FIFO, rx_buffer);
    else {
      hpam_save_area->in_q_tail++;
      *(hpam_in_q) = *(hpam_in_q)-1;
    }

  release:
    *hpam_busy=MED_RX_FREE;   /* release this buffer */
    goto start;
  } /* end if anything in the card */
  /* timeout and retransmission check */

  if (hpam_check_buf ==0) {
    hpam_check_proc= (hpam_check_proc+1) % HPAM_partition_size;

    if (hpam_check_proc == HPAM_self_address)
      hpam_check_proc= (hpam_check_proc+1) % HPAM_partition_size;
  }

  hpam_check_buf = (hpam_check_buf == (MAXDEPTH-1)) ? 0 : hpam_check_buf+1 ;

  if ((hpam_req_state[hpam_check_proc][hpam_check_buf].state&TX_RD)==0) {
    timecheck(&(hpam_req_state[hpam_check_proc][hpam_check_buf]));
  }


}   /* HPAM_poll */
```