PPE-level Protocols for Carpet Clusters[1]

Mark R. Swanson
Leigh B. Stoller

UUCS-94-013

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

April 11, 1994

Abstract

We describe the lowest level of a suite of protocols for workstation cluster multicomputers: the parts implemented in hardware by a Protocol Processing Engine (PPE) and the software level immediately above the PPE. The stated goal of this work is extremely low end-to-end latency communications on independent workstations connected by a packet switching communication fabric. The workstations are expected to run a commercial operating system and must present the same security characteristics as traditional protocols. We begin with a realization of *sender-based* protocols. Such protocols can avoid much of the copying that slows down traditional approaches and can also reduce the overhead involved in demultiplexing packet streams and notification of recipients. Finally, we present some measurements of an early prototype.

1 Introduction

We describe the lowest level of protocols for workstation cluster multicomputers: the parts implemented by the Protocol Processing Engine (PPE) and the software level immediately above the PPE. We maintain a sender-based[2] flavor, but have sought to simplify mechanisms at this level wherever possible. We seek primarily to determine an appropriate (economically implementable) level of functionality for the PPE.

2 Overview

The basic design goals are, of course, motherhood and apple pie: low latency, high throughput, and low impact on the host resources–both cpu cycles and bus capacity. The specific areas we intend to address to achieve these goals are several:

[1]This work was supported in part by a contract from Hewlett Packard Laborities.

1. avoiding extraneous copies of the data;
2. relying on hardware support (DMA) to do the unavoidable copy whenever it is profitable;
3. providing a **direct** interface with low *total* overhead for cases where DMA is actually uneconomical or inappropriate;
4. relying on hardware to perform packetization and reassembly of large messages;
5. minimizing demultiplexing and dispatching overheads and moving some portion of them into hardware;
6. avoiding context switches when the *expected* latency of a remote transaction is less than the cost imposed by a context switch;
7. reducing the number of interrupts needed to continue transactions that do result in context switches.
8. minimizing operating system costs for common operations.
9. taking advantage of the characteristics of the interconnect.

In specifying these activities, two constraints must also be satisfied:

1. minimizing the complexity of required hardware support;
2. maintaining acceptable levels of security.

2.1 Avoiding Copies

In general, to avoid unnecessary copy operations, a protocol must allow for the **data** to be sent directly from user memory and to be received directly into user memory. For the receive operation, therefore, it must be possible for the device driver (or hardware) to determine from the (packets of a) message where the data must go.

Practically speaking, avoiding copies for received messages means that receive buffers must be wired, since otherwise the possibility of page faulting arises. Page faulting involves significant latency, on the order of milliseconds, with one of three possible outcomes:

1. stalling the receipt of new packets;
2. introducing intermediate buffering, which implies the need for undesirable copying;
3. dropping packets which are (temporarily) undeliverable.

None of these is desirable; none of them fits within the design goals. Therefore we accept the need for wired receive buffers. It is also likely that the PPE will be provided only with physical, rather than virtual, addresses for receive buffers. Address translation hardware would only complicate the PPE, making it either more expensive or slower, and adding no real value, since the virtual pages will be backed by wired physical pages.

Wiring of send buffers is not strictly necessary for *processor mediated* or *direct* transfers, wherein the processor reads each word of the message and writes it to the device. The consequences of page faults are less dire–as long as the software ensures that no page faults can occur within a single packet transmission, only the sending process and possibly its receiver are affected by an increase in latency. The device, as we propose it, should be available for other processes to send messages/packets in the meantime.

2.2 Hardware Copying

The cost function for message sending is complex, including host resource components, specifically cpu cycles and memory bus bandwidth, as well as controller latency and interconnect bandwidth components. We expect that providing two modes of operation will allow optimization of selectable components of the cost function based on application and system needs. The processor mediated, or direct IO (DIO), mode already has been touched on briefly. A DMA mode is also envisioned for cases where cpu cycles can be better spent on work other than message sending and where latency is less of an issue. Since no VM hardware is envisioned for the PPE, DMA-type transfers will require wired send buffers and physical addresses.

In message receipt, the unavoidable copying is *always* performed by the PPE hardware, as mentioned above.

2.3 Packetization and Reassembly

Packetization is a relatively simple task, being inherently serial; it imposes no need for additional copy operations and entails the maintenance of only a little state. One difficulty arises in the event that the device stops accepting packets due to congestion in the fabric or delays in the receiving node. In our design, a heuristic will need to be devised to determine when the transfer should be deemed "stalled" and the DMA engine must be freed for other messages.

Packet arrival is asynchronous, making reassembly an event-driven process. The events also tend to be "small-grained", each involving a single packet of relatively small size. Utilizing an interrupt and/or context switch for each packet arrival would consume significant host resources and would result in high per-packet latencies. This is in contrast to the sender, which can produce a stream of packets without high-cost interrupts or context switches; such a system would be inherently unbalanced. The expected nature of the interconnect also renders reassembly more complex than packetization. Packets can arrive out of order and packets from several messages can be interleaved.

We view efficient reassembly as a very critical process for achieving low latency and providing balance within the system. Hardware support must be adequate to eliminate the need for packet receipt interrupts, to establish the appropriate context for each packet's delivery, to determine message completion, and, in general, to overcome the inherent speed "advantages" of senders.

2.4 Demultiplexing and Dispatching

One way in which interface hardware can help lower communication costs is by interpreting simple in-host routing directives to deliver notifications (see Section 3.3) of message arrival to the appropriate process. Each message will carry such information, indirectly specified by data in its

constituent packet's headers. The mechanisms we describe also provide the application with the flexibility of focusing all message arrival events into one notification area or spreading them across several such areas.

When the PPE is requested to interrupt the cpu for message arrival, the information necessary to service that event will be made directly available to the kernel. The kernel will not be required to perform searches or to interpret packet headers to determine which process(es) need be awakened to process the message.

2.5 Context Switch Minimization

We provide mechanisms to allow the process flexibility in dealing with communication events. Message arrival interrupts are selectable on a connection basis and can be dynamically enabled or disabled (via kernel mode code). We expect that the choice of polling vs. kernel-supported waiting for communication events will be a tuning process and should not be pre-empted by the PPE design.

2.6 Minimizing OS Costs for Common Operations

The most common operations, and therefore the most performance critical ones, are message transmission and reception. For these operations, we propose special lightweight system calls rather than user access to the interface. We take this approach for the following reasons:

- Latency for small messages will be dominated by bus transaction times and these times are essentially the same for user mode approaches as for kernel mode approaches.
- Latency for large messages is dominated by bandwidth (memory and/or interconnect) and will be DMA driven; the details of user vs. kernel mode IO will not materially affect performance.
- User space IO *requires* significant protection support from the hardware, while such protection can easily (and efficiently) be provided by kernel mode software. This is an interface complexity argument.

Uncommon operations, such as connection establishment and buffer management are expected to be infrequent. They will likely be handled via traditional system interfaces (though the system calls will necessarily be new).

2.7 Hardware Complexity

The hardware support provided by the PPE must itself introduce minimal latency and impose only reasonable cost. Designs employing complete cpu's are unlikely to perform adequately at reasonable cost. Protocol operations in the hardware requiring searches and complex decision making are ruled out.

Significant savings in bus transactions (especially in the face of bus contention) may be realized by providing memory on the PPE itself. The cost of sufficiently fast memory will likely dictate that only modest amounts of store will be available on the PPE.

2.8 Maintaining Security

The desired level of security is essentially that of current networking implementations in Unix-based systems. User code should not be able to compromise the local system or a remote system by use of these protocols. A user process should not be able to compromise other processes, local or remote, with which it is not in direct communication via these protocols. The extent to which a user process can compromise a process with which it does communicate via these protocols will be limited to two kinds of attack:

1. a sender may cause messages to be written anywhere within the bounds of the receiver specified buffer. Our protocols do not constrain the sender's access within this region.
2. a process may fail to adhere to a higher level protocol based on these protocols, leading to faults in the other processes it communicates with.

We view these as application programming errors which are beyond the scope of our efforts to resolve.

2.9 Capitalizing on Interconnect Characteristics

Where the interconnect provides functionality normally provided by higher-level protocols, we should take advantage of it. This includes features such as the following:

- flow control within the fabric;
- guaranteed packet delivery;
- hardware generated and checked checksums;
- uniformity of the interconnect.

3 Operational Description

Having described the general characteristics of our approach, we now present an operational description, that is, we detail the steps involved in sending a message, in receiving a message and in processing a message. The required data structures and hardware support for each activity are introduced as we go along. Connection establishment protocols will be addressed in a separate document.

3.1 Message Transmission

The message transmission process begins with the transition of the user thread into supervisor state[2]. The virtual memory context remains that of the user process, augmented with access to kernel data structures and to the PPE device in IO space. The user call must include:

[2]In our testbed system which is Hewlett Packard PA-RISC-based, this is accomplished via a `gate` instruction. The implementation on other systems will differ, of course.

```
struct sslot {
        short               remote_node;
        short               this_node;
        short               remote_slot;
        struct _s_control {
                u_short         interrupt     :1,
                                busy          :1,
                                reserved      :5,
                                type          :2;
                                incarnation   :8,
        } control;
        unsigned            remote_offset;
        unsigned            buffer_base_phys;
        unsigned            buffer_base_virt;
        unsigned            buffer_size;
        unsigned            receiver_link;
        unsigned            completion_flag;
}
```

Figure 1: The sending slot.

- a logical connection identifier;
- the offset, within the send buffer, of the data to send;
- the size of the data to send;
- the offset within the receiver's buffer where the data will be placed;
- some flag or option values.

3.1.1 Validating the Arguments

The connection identifier is process-specific; it is range checked and then used to locate a send slot (*sslot*; see Figure 1) which contains the actual connection description. Send slots are software-only data structures; the PPE need not be aware of their structure. The fields of a send slot are defined as follows:

- remote_node - fabric address of the destination node.
- this_node - fabric address of the source node.
- remote_slot - index of the target rslot on the receiving node.
- control - flags and and other control bits:

  - interrupt - If set, the PPE interrupts the CPU when the message transmission completes.
  - busy - the state of the connection dictates that no transmissions be sent via the sslot.
  - reserved
  - type - message (or packet) type.
  - incarnation number (see Section 4.1).

- remote_offset - offset within the receiver's buffer at which the sender can start placing messages.
- buffer_base_phys - physical address of the associated send buffer.
- buffer_base_virt - virtual address of the associated send buffer.
- buffer_size - size in bytes of the send buffer.
- receiver_link - link to the corresponding receive slot.
- completion_flag - a location the PPE can mark to indicate completion of a transmission.

The `sslot` contains a `busy` bit, which can be used by the software to mark the *connection* using the sslot as busy. For example, when the request part of a *split-phase* RPC[3] is sent, the associated sslot can be marked busy until the reply is received; alternatively,the busy bit can be cleared when the message is known to have arrived at the destination node. Once the send slot is validated by the kernel, the offset supplied by the user is compared to `buffer_size`; if the offset is greater, the call returns with an error. The sum of the user-provided size and offset is also compared to the `buffer_size`; if the sum is greater, an error is returned. It is desirable to perform range checking on the remote buffer at this point. This entails obtaining the remote buffer size at connection establishment and making it part of the `sslot`. This size is compared against the sum of the message size and remote offset provided by the user. The benefit lies in earlier fault detection, since it is assumed the receiving PPE would still perform such a range check to contain even erroneous writes to only the receiving node's memory to specified buffer areas.

3.1.2 Initiating the Transmission

The PPE should present a small number (we currently envision four) of *send_descriptors* (see Figure 2) which are used by the kernel to initiate and control message transmission. Each one contains fields for routing and remote message placement information to form packet headers, and each also contains the information to drive DMA for PPE-controlled packetization:

- status - control and status register:
  - busy - the PPE is actively using the send_descriptor to transmit a packet;
  - stalled - the ongoing send has been stalled by the fabric for flow control reasons;

[3]By *split-phase* RPC we mean that the calling process chooses not to block waiting for the reply. Control returns to the process, which can perform other work and later explicitly check for the expected reply.

```
struct send_descriptor {
        struct sd_status {
                unsigned        busy       :1,
                                stalled    :1,
                                done       :1,
                                in_use     :1,
                                interrupt  :1,
                                completion :1,
                                direct_io  :1,
                                go         :1;
        } status;
        short                   bias;
        short                   dst_node;
        short                   src_node;
        short                   dst_slot;
        struct _pkt_control     control;
        unsigned                remote_offset;
        unsigned                *local_address;
        unsigned                msg_size;
        unsigned                *completion_address;
        unsigned                pkt_buffer[PKT_SIZE];
}
```

Figure 2: A PPE send descriptor.

- done - the PPE has finished sending all packets specified by the send_descriptor;
- in_use - a process is preparing the send_descriptor for a message transmission or a transmission is in progress;
- interrupt - the PPE is instructed to issue an interrupt when it sets the done bit;
- completion - when set, the PPE should write the sd_status word to the location specified by the completion_address field of the send_descriptor (before posting the interrupt if **interrupt** is set).
- direct_io - the PPE should take data from the send_descriptor **pkt_buffer** rather than doing DMA;
- go - when set, the PPE is expected to use the send_descriptor for transmitting; when not set, the PPE should ignore the send_descriptor.

- bias - typically the count of packets in a message.
- remote_node - fabric address of the destination node.
- this_node - fabric address of the source node.
- control - see _pkt_control in Figure 3.

- remote_offset - offset within the receiving buffer where the message will be placed.
- local_address - physical address used by the DMA engine to acquire the message data.
- msg_size - size in bytes of the message.
- completion_address - address of a location to which the PPE can write ¡completion status.
- pkt_buffer - a buffer large enough to hold an entire packet's payload; it may not be present in all send_descriptors.

Initiating a transmission involves finding a free send_descriptor, loading it with the appropriate values, and telling the PPE to proceed. The kernel turns off interrupts for the duration of this process, which is expected to be very short (see Section 7 for current measurements).

Finding a free send_descriptor entails reading the status field for each send_descriptor and testing the **in_use** bit. Acquiring the send_descriptor entails setting **in_use**. Since interrupts are off, setting **in_use** can be delayed until a write to the status register is required. In the event that no free send_descriptor is available, interrupts are re-enabled and control passes to the "busy PPE" handler (see Section 3.1.6). Having found a free send_descriptor, the kernel performs the following steps:

1. it reads the first two words of the **sslot** and writes them to the first two words of the send_descriptor.
2. it reads **remote_offset** from the **sslot**, adds the remote offset supplied by the call, and writes the sum to the send_descriptor **remote_offset**.
3. it forms the local physical address for DMA from the offset supplied in the call and **buffer_base_phys** from the sslot. It writes this to the send_descriptor **local_address**.
4. it computes the number of packets in the message and writes it to the send_descriptor **bias**.
5. it writes the size from the call to the send_descriptor **msg_size**.
6. for some systems, it will flush the message contents from the data cache to ensure consistency with main memory.
7. it writes a control word to the send_descriptor **status** which sets **go** and **in_use** (and possibly **interrupt**).
8. it re-enables interrupts.

The format of the initial two words of the sslot and send_descriptor is very fabric dependent; we have tailored them for the proposed R2[1] fabric packet header (see Figure 3). The fields of the packet header are as follows:

- dst_node - the fabric address of the destination node.
- src_node - the fabric address of the source node.

- dst_slot - the receive slot the message is intended for.
- - control (and status) bits:
  - use_msg_size - causes receiving PPE to remember the msg_size from this packet.
  - use_msg_offset - causes receiving PPE to remember the msg_offset from this packet.
  - use_bias - causes the receiving PPE to use the bias from this packet.
  - reserved
  - type - message (or packet) type.
  - incarnation number (see Section 4.1).
- offset - offset of the message within the receive buffer.
- pkt_size - the size, in bytes, of the packet (to allow short packets).
- bias - typically the number of packets in the message.
- msg_size - the size of the message in bytes.
- hdr_checksum - hardware generated checksum of the packet header.

An alternative design that sought greater portability across fabrics at the cost of PPE complexity would involve storing a table of connection information or preformatted packet headers on the PPE and supplying the PPE with an index into this table via the send_descriptor rather than the supplying the actual routing information.

Normally, **use_msg_size**, **use_msg_offset**, and **use_bias** are all set in the control word. They are used by the receiving PPE; this is covered in detail in Section 3.2.

When using DMA, the size of all packets but the last is fixed at **PKT_SIZE** bytes, where PKT_size is a fabric dependent parameter; the final packet may be short. The number of packets is simply **(msg_size + PKT_SIZE - 1) / PKT_SIZE**; more complex message models (see Section 3.1.4) compute this value differently.

The PPE forms packets sequentially from the data described by the send_descriptor. As each packet is sent, the PPE increments **remote_offset** in the send_descriptor. The current value of **remote_offset** is placed in each packet header to direct the receiving PPE in depositing the packet's data. **use_msg_size**, **use_msg_offset**, and **use_bias bits** in the send_descriptor **control** are zeroed by the PPE after the initial (and each following) packet is sent. That is, only the initial packet sent in a DMA-processed message will have these bits set in its header.

3.1.3 Initiating Direct IO

A direct interface (DIO) is also supported for messages comprising a single packet. All the validations and send_descriptor setup of the DMA case are performed, with the following exceptions:

- the kernel does not set **local_address** in the send_descriptor, but copies packet data directly into the send_descriptor **pkt_buffer**;

```
struct pkt_hdr {
        unsigned short      dst_node;
        unsigned short      src_node;
        unsigned short      dst_slot;
        struct _pkt_control {
                unsigned        use_msg_size   :1,
                                use_msg_offset :1,
                                use_bias       :1,
                                reserve        :3,
                                type           :2;
                                incarnation    :8,
        } control;
        unsigned            offset;
        unsigned            msg_size;
        unsigned short      bias;
        unsigned short      pkt_size;
        unsigned short      reply_slot;
        unsigned short      hdr_checksum;
}
```

Figure 3: The R2 packet header.

- the kernel never needs to flush the message contents from the data cache;
- the kernel sets `direct_io` in the status word it writes to the send_descriptor.

[It is an open question at present whether all send_descriptors should have a packet buffer, i.e., can be used for DIO, or if just a subset are so equipped. Either a status bit, `dio_capable` can be provided, or the capable send_descriptor or set of send_descriptors will be statically known to the kernel.] For fabrics with large packet sizes, it *may* be desirable to re-enable interrupts before copying the data. The `in_use` bit could then be used to reserve the send_descriptor and allow safe re-enabling of interrupts.

3.1.4 Initiating Complex Message Transmission

The messages described so far have been "simple". They are comprised of contiguous packets and the message length is known before the send is initiated. There are situations where these conditions may not hold. One possibility is a message that needs to scatter blocks of data throughout the receiver's buffer. Another is the case where a stream of data is sent via DIO and its total length is not known until the end of the stream is reached (this could occur, for example, in an application that attempts to marshal arguments for an RPC directly into the fabric rather than into an intermediate buffer).

For a "scatter" message, the transmission of the message will likely comprise a number of DMA-mediated transmissions. The kernel would compute the total number of packets involved and write that number to `bias` in the send_descriptor for exactly one of the transmissions, as well

as setting **use_bias** for that same transmission. It is not clear what message offset, if any, would be appropriate for notifying the receiving process of a "scattered" message; it may either be left unspecified or an interface to higher level protocol code will be needed to allow specification of the offset. A similar situation obtains for the message size. Since out-of order delivery of the packets of a message is an accepted characteristics of the underlying fabric, a "scatter" message could make concurrent use of multiple send descriptors. While the PPE sends one portion of the message using one send descriptor, the kernel could be initiating the send of another portion using a different send descriptor. This would allow pipelining of the setup and transmission times of consecutive message portions.

For a "stream" message, a series of DIO packets would be used. The message offset of the first packet would be the one needed for notifying the receiving process; hence, **use_msg_offset** would be set in the send_descriptor for the first packet. The message size and number of packets would not be known until the last packet was ready to go. **use_msg_size** and **use_bias** would be set for this last packet and **msg_size** and **bias** in the send_descriptor would be set appropriately.

3.1.5 Controlling Transmission

Once the transmission is initiated, subsequent behavior on the sender's part is largely application dependent. The send call includes an options argument which specifies the desired behavior. If the send was the request part of a synchronous request/reply pair (e.g., the call part of an RPC), the sender may block waiting for the reply (see Section 3.2), possibly setting up some timeout mechanism as well. We avoid discussion of initiating the timeout mechanism in this case, since it can be performed after the transmission is initiated and thus does not add directly to the transmission latency.

The send may also be non-blocking, either since it is a reply itself and no response to it is expected, or because the sender wishes to overlap other work with the transmission and remote message service time. In either case, the sender may need to be able to determine when the transmission has completed, possibly to allow reuse of the send buffer space. For short messages, the sender could simply spin until **done** is set in the send_descriptor status field. This must be done atomically along with ensuring that the send_descriptor still "belongs" to the desired transmission. Since interrupts will have been re-enabled, the send_descriptor could well be in use by another process. One way to identify the transmission described by a send_descriptor is by the value stored in its **completion_address**. Uniqueness can be assured by using a wired physical address within the sender's space. The **completion_address** can also be used to specify a location where the PPE should write the **status** of the send_descriptor when a message transmission completes. This behavior is enabled by setting **completion** in the send_descriptor **status** when the transmission is started.

3.1.6 Dealing with a Busy PPE

A number of conditions could result in a busy PPE, i.e., all of the send_descriptors of the PPE being marked in use and not yet done. An obvious cause would be the initiation of concurrent very long transfers by one or more processes. Another case arises when one or more send_descriptors are in use but are not "making progress". Flow control within the fabric can cause this situation

to arise if the receiver is slow in accepting packets or has actually stopped accepting packets. The **stalled** bit in the send_descriptor **status** will be set each time the PPE fails, due to flow control, in an attempt to send a packet using the send_descriptor. It is reset on a successful attempt. The **remote_offset** in the send_descriptor is also an indicator of progress. If it does not change in a "reasonable" amount of time, the software may conclude that the transfer is being subjected to flow control. The software may set **busy** in the **status** of the send_descriptor and then proceed to "unload" the state of the send_descriptor, making it available for other transmissions. The software is responsible for continuing the unloaded transmission at a later time. In the event that the destination had actually gone down, the sending node will *eventually* be informed and the transmission terminated and an error status returned at the **completion_flag** location, if it was provided. We expect that the **completion_flag** address will always be supplied for this reason, but that the **completion** bit in the send_descriptor will only be set for non-polling senders. A polling process may decide to become a non-polling process by setting **completion** in the send_descriptor.

3.2 Message Reception

Message reception occurs asynchronously. Each packet contains sufficient information to enable the PPE to deposit its payload directly into memory. In particular, each packet's header contains an *rslot* index which identifies the receive slot (see Figure 4) to be used in processing the packet. The fields of the receive slot structure are as follows:

- packets_to_go - used by the PPE to determine when a message is complete.
- msg_size - extracted from any packet header with **use_msg_size** set; copied into the notification entry on message completion.
- msg_offset - extracted from any packet header with **use_msg_offset** set; copied into the notification entry on message completion.
- control - Various state and control bits:
  - valid - when set, the PPE can use the slot for packet delivery.
  - indirect - specifies that the **buffer_base_phys** actually points to a page map of the buffer.
  - incarnation number (see Section 4.1).
  - type - message (or packet) type.
- buffer_base_phys - the physical address of the receive buffer or of a page map for the receive buffer.
- buffer_size - the total size of the buffer.
- notes - physical address of a notification list; see Figure 5.
- serviceid - an identifier supplied to the receiving process.
- nextfree - index of next free slot entry. Used only by the processor to maintain a freelist.

```
struct rslot {
        struct {
                unsigned        packets_to_go;
                unsigned        msg_size;
                unsigned        msg_offset;
        } ppe_write;
        struct {
                unsigned        buffer_base_phys;
                unsigned        buffer_size;
                notelist_t      *notes;
                struct _r_control {
                        u_short valid       :1,
                                indirect    :1,
                                reserve     :4,
                                type        :2;
                                incarnation :8,
                } control;
        } ppe_read;
        struct {
                short           nextfree;
                short           serviceid;
                unsigned        buffer_base_virt;
        } cpu_only;
}
```

Figure 4: The receive slot.

- buffer_base_virt - the virtual address of the receive buffer.

A table containing the PPE-specific portions of the rslot may be implemented in memory on the PPE. Since each packet arrival requires access to an rlsot entry, significant bus traffic could be generated if the tables reside in main memory; in addition, the potential for delay in gaining access to main memory could increase reception latency. An alternative approach for reducing these costs would be to provide some form of caching on the PPE for recently-used rslots. Such a caching strategy would modestly complicate the following sequence. On packet arrival, the PPE performs the following steps:

1. it checks the destination node; if it is not for the local node, the packet is an error packet.
2. it range checks the rslot index (**dst_slot** in the packet header); if it is out of range, the packet is an error packet.
3. it reads the PPE portion of the rslot selected by **dst_slot** into a buffer; future references to "rslot" imply this buffered copy;
4. it checks whether **valid** is set in the rlsot; if it is not set, the packet is an error packet.
5. it checks whether **incarnation** in the packet header is equal to **incarnation** in the rslot; if they are not equal, it is an error packet.
6. **offset**, and the sum of **offset** and **pkt_size**, from the packet header are both range checked against **buffer_size** in the rslot; if the sum is out of range, the packet is an error packet.
7. it copies the packet data into main memory at the location formed by the sum of **buffer_base_phys** from the rslot and **offset** from the packet header; the size of the payload is specified by **pkt_size** in the packet header.
8. if **use_msg_size** is set in the packet header, **msg_size** from the packet header is stored into the **msg_size** in the rslot;
9. if **use_msg_offset** is set in the packet header, **offset** from the packet header is stored into the **msg_offset** in the rslot;
10. if **use_msg_bias** is set in the packet header, **bias** from the packet header is added to **packets_to_go** of the rslot.
11. **packets_to_go** in the rlsot is decremented; if it becomes zero, a complete message has arrived and notification action must be taken (see Section 3.3);
12. **packets_to_go**, **msg_size**, and **msg_offset** of the buffered rslot are written back to the rslot entry in the table.

```
typedef struct notelist {
        lock_t          lock;
        unsigned        ref_count;
        unsigned        head;
        unsigned        tail;
        unsigned        mask;
        unsigned        size;
        unsigned        interrupt;
        unsigned        pad;
} *notelist_t;
```

Fields:

- lock - protect against concurrent updates by PPE and cpu.
- ref_count - count of rslots pointing at this notelist.
- head - head pointer index into the notes array.
- tail - tail pointer index into the notes array.
- mask - log2 of the notes array size for wrap around.
- size - size of the array; must be a power of 2.
- interrupt - if non-zero, the PPE will generate an interrupt after writing the notification.
- notes - variable length array of note_t structures.

Figure 5: The notification list structure.

3.3 Receiver Notification

When a complete message has been received, the PPE uses the **notes** of the rslot to retrieve the notification list (see Figure 5) header and subsequently write a notification (see Figure 6). If **interrupt** in the notelist structure is non-zero, the PPE posts an interrupt after writing the notification. Multiple rslots may point to the same notification list; this allows a process to monitor a single list for requests arriving on multiple rslots. Since the notification list is shared between the PPE and the cpu, a lock is provided to protect access to the list header. This is only strictly required if the head and tail indices occupy the same cache line, as they do not share any data in a write-write fashion. The list is actually represented as a circular queue. On some systems, the kernel must flush the notelist from the data cache after any modification to ensure that the PPE sees up-to-date values. Likewise, these systems may require that the kernel purge the notelist from cache before accessing it, to ensure that the kernel sees the correct value of the **tail** field.

The formation of the actual notification uses information from both the current packet (the one that triggered notification) and the buffered rlsot used in processing that packet. **msg_offset** and **msg_size** in the notification come from the rslot; the other fields come from the packet header.

The PPE checks for a full notification queue; if the queue is not full, it will write the notification into the entry at index **last** ANDed with **mask**, increment **last**, and write **last** back to the header. If the queue is full, the notification will be inserted into the notification queue reserved for error

packets (see Section 6); in this case, `notefull` in the notification will be set. The process handling errors (probably the OS) may allow the application to recover in this case, since the message has been delivered; only the notification was undeliverable.

This notification structure supports individual rslots adequately: some cpu-accessible structure must be available for notification information, and it must be possible for the cpu to indicate that the notification has been consumed. If connections were limited to a *single message in process*[4], a single entry within the rslot would be sufficient. When the receiver finished processing the notification, it could inform the PPE that the entry was once again free.

More importantly, this structure will support two more complex, but important, cases: streams that are likely to have multiple incoming messages (like the error packet stream) and processes that need an efficient means to listen to multiple slots. For enhanced generality, we have thus separated the actual notification structure from the rslot.

4 Connection Establishment

In this section we will discuss the information flow required at connection setup, but not the entire connection establishment protocol. At the lowest level, (one-way) connection establishment entails these actions:

- allocating an `rlsot`;
- associating buffer space with that `rslot`;
- associating a notification list with that `rslot`;
- setting the control field of that `rlsot`;
- communicating connection information to the sending node/process.

The connection information conveyed to the sender includes the `rslot` index, the receiving node's current incarnation number (see Section 4.1), an offset within the `rlsot`'s buffer at which the sender can deposit data, and the size of the (portion of the) buffer associated with the connection being set up. The `offset` within the receiver's buffer is available to provide disjoint sub-buffers in the event that multiple senders are allowed to transmit to a single rslot.

4.1 Incarnation Numbers

In order to tolerate the loss and subsequent reappearance of a node, it is necessary to guard against packets that are addressed to slots making up connections that disappeared in the node crash. These packets might be sent from nodes holding connections to the rebooted node at the time it went down. When such a packet arrives at the rebooted node, use of the indicated rslot by the PPE would constitute an error, since the connection it was part of no longer exists. The rslot might simply be marked invalid, in which case the PPE would treat it as an error packet. The rslot might

[4]This is a higher level protocol constraint, in contrast to *single message in flight*, which is a PPE-level constraint.

```
struct notification {
        short                   dst_node;
        short                   src_node;
        short                   dst_slot;
        short                   reply_slot;
        struct _note_control {
               notefull     :1,
               reserve      :5,
               type         :2;
               incarnation  :8,
         } control;
        short                   unused[7];
        unsigned                msg_offset;
        unsigned                msg_size;
}
```

Fields:

- dst_node - except for error packets, always the current node;
- src_node - used in error handling to return error messages;
- dst_slot - used by the kernel to identify the connection on which the message arrived;
- control - Various state and control bits:
  - notefull - set by the PPE when it fails to find space in the notification list associated with the rslot specified by `dst_slot`.
  - incarnation number (see Section 4.1).
  - type - message (or packet) type.
- msg_offset - returned to the application;
- msg_size - returned to the application.

Figure 6: The individual notification entry structure.

also be valid, however, as part of a new connection. The PPE must have a mechanism to detect this case.

A simple approach would entail a rebooting node broadcasting to all other nodes the fact that it is rebooting. The node would need to wait for replies to the broadcast to be certain that no stale connections remained. Such a global operation has poor scaling characteristics for large systems, though we do allow the possibility of limited use of broadcast below.

We have addressed the rebooting node issue by adding an *incarnation number* to the control field of the packet header (see Figure 3). This incarnation number is initialized when the connection is established. The node holding the receive slot passes back its value of the incarnation to the node holding the send slot. The incarnation number is thus stored in the control field of both the send (see Figure 1) and the receive slot (see Figure 4), and is passed in the packet header for each

packet sent. When a packet comes in, the PPE compares the incarnation number in the packet header to the incarnation number in the receive slot. If the numbers do not match, the packet is ejected as an error (see Section 6).

When a node reboots, it uses an incarnation number that is stored on disk, and writes a new, incremented value back. This value is passed back during each connection setup in which the node is the receive side of the connection. Since the value is only 8 bits wide, it is *possible* that a machine could reboot enough times for the counter to wrap around, without a message ever being sent from a node holding a stale connection. The result would be an incarnation number that looks correct, but is not. One possible algorithm to guard against this is as follows: whenever a node reboots and the new incarnation number written to disk will be 0, i.e., the counter has wrapped around, the node sends a message containing its new incarnation to all other nodes. Only when all the nodes have responded, or are known to have failed (see Section 6), does the node write out the new incarnation number to disk and establish new connections. If the node should fail before writing out the new incarnation number, it will simply repeat the broadcast when it reboots; the only costs are in message traffic, the delay before the rebooting node can establish new connections, and the receiving nodes' time spent processing the incarnation number messages. This processing involves scanning the `sslots`, marking any which have the rebooting node as the destination as invalid. The cost in messages is linear in the number of nodes, and the messages will be quite small.

5 Booting

We currently envision having two rslots for each node in a "system" on each node in the system dedicated to inter-node kernel communication. That is, an rslot per node for incoming requests and an rslot per node for replies. This is an artifact of our model of point-to-point connections. For carpet clusters of modest size (up to several hundred nodes) this should not be prohibitively expensive. For larger configurations, a more complex software protocol (many-to-one, at most one packet per message connections) could reduce this requirement to two rslots total per node for inter-kernel communications. We defer consideration of this more complex model, noting that we believe the specifications above will provide adequate support should we find it necessary to develop the specialized kernel-to-kernel protocols.

6 Errors

Error packets represent a special case for the PPE. The slot number in the error packet cannot be used to select an `rslot` – if the error packet was just ejected by the fabric onto the node, it will likely have been destined for some other node. If the packet attains error status within the PPE, it is likely because the `rslot` cannot receive the packet for some reason. The current design entails writing a notification entry to the notification list associated with a reserved `rslot` (e.g., rslot 0). The payload will be discarded. The notification includes sufficient information that the error handling process (likely the OS) can take appropriate action.

As an example of "appropriate action", consider the case where a node goes down and there are packets enroute to it. The packets will eventually be ejected by the fabric to some node(s). The destination node in the packet will not match the receiving node, thus the PPE will place

a notification entry onto the reserved error slot's notification list and post an interrupt if the notification list's `interrupt` field is non-zero (which will probably be the case). As stated above, we currently envision discarding the packet body.

The kernel, at interrupt level, will send a priority packet to the source node indicating that the original destination node is probably down; the body of this packet will be the original notification. The kernel may optionally remember that it has sent a crash notification to the source node. Subsequent error packets with the same destination/source pair could then simply be discarded. The kernel will mark any local connections with the crashed node as destination to prevent further local use of those connections. The kernel will also note that the crashed node is just that–crashed, affecting future attempts to connect to that node.

The source node should mark the affected connections to reflect the loss of the destination. It should, of course, send no further messages using these stale connections.

Other example errors include packets with bad incarnation numbers and packets addressed to invalid rslots (both indicating a stale connection), packets which fail the buffer range checks (should never happen!), packets with bad header checksums (should also never happen), and packets destined for rslots with full notification lists.

7 Performance

The target systems for which the R2 and PPE are intended lie in the future. The performance characteristics of these systems will differ markedly from the systems we are developing on. Assuming that the processor architecture remains essentially the same, we have implemented prototypes of the send and receive "lightweight" system calls and a simple RPC built on top of them. The prototypes were run on HP 720's with Medusa FDDI controllers as the interconnect.

The basic send path from the start of the system call to the initiation of the message sending by the Medusa comprises 70 instructions. Of these, 8 are measurement-related and 24 are Medusa specific. We expect the 24 Medusa specific instructions to be replaced by 6 PPE-specific instructions when the PPE becomes available, yielding a code path length of 44 instructions to do a 1 word DIO send. A DMA send follows essentially the same path (and has essentially the same cost). The average CPI (measured) for the 70 instructions was in the range of 3 to 3.5. This includes accesses to four data cache lines and 1 read and 7 writes to IO space (the Medusa).

We expect that on future systems, the cache miss penalties will be larger as will the expense of IO space accesses, while at the same time absolute times will decrease.

RPC times (1 word of data), user to user, were measured at 120 microseconds. The receive side was in a kernel polling loop inside a system call; if a context switch had been required instead, we would expect to add 20 to 25 microseconds on each end for a total of 170 microseconds. Of the total, 18 microseconds on each end was spent in an FDDI interrupt, 12 microseconds on each end in an PPE packet receipt emulation, 5 microseconds in our protocol send path, and 11 microseconds crawling out of the kernel system call for receive.

8 Conclusion

We have described low-level protocols and a communications fabric interface for carpet clusters. The combination implements a sender-based protocol and, based on our prototype, will achieve the low latency and high bandwidth necessary for effective multicomputing with such clusters.

References

[1] DAVIS, A., CHERKASOVA, L., KOTOV, V., ROBINSON, I., AND ROKICKI, T. R2 - a damped adaptive multiprocessor interconnection component. In *Proceedings of the University of Washington Conference on Multiprocessor Interconnects* (May 1994).

[2] WILKES, J. Hamlyn - an interface for sender-based communication. Tech. Rep. HPL-OSR-92-13, Hewlett-Packard Research Laboratory, November 1992.