Hamlyn — an interface for sender-based communications

John Wilkes

HPL–OSR–92–13, 30 November 1992

Hewlett-Packard Laboratories
Operating Systems
Research Department

Approved for external distribution

Keywords: IPC protocols, sender-based communication

Providing high performance, low-overhead inter-processor communication is a necessity if the promise of scalable distributed-memory multiprocessors is to be achieved. This paper uses a characterization of three different types of interconnect traffic to drive the development of an innovative high-speed interconnect interface. This uses sender-controlled message placement at the recipient, which has the effect of greatly reducing the cost and complexity of message handling.

The contributions of this work are in (a) elucidating the traffic model; (b) in defining the sender-driven communication scheme; and (c) in the detailed description of an efficient, protected interface to the interconnect hardware that allows applications running in non-privileged mode to access the interconnect directly, without operating system intervention.

This version of the paper contains a complete high-level design for the first version of Hamlyn—a hardware interface that accommodates all the Hamlyn functionality. Future work on the protocol stacks and implementation work will doubtless improve and modify this interface. Until then, this description serves as a functionally complete snapshot of the Hamlyn approach.

1 Background

A distributed-memory multi- or parallel-processor, variously called a multicomputer, "shared nothing machine", or "DMPP", is composed of a set of processing *nodes* (processors and local RAM) with no memory physically shared between the nodes (Figure 1). A high-speed interconnect fabric supports communication between the nodes. The interface to this fabric is the subject of this paper.

The main contribution of this paper is a new scheme for managing memory-to-memory communication: one in which the sender determines the address at which a message will arrive in the receiver's memory. A complete set of the mechanisms necessary to achieve this are described, together with a short description of the software protocols that could be layered on top of it.

1.1 Multicomputer properties

A single node in a multicomputer may be a uniprocessor or a shared-memory multiprocessor in its own right. The nodes may be homogeneous or heterogeneous, although for simplicity this work assume that the nodes use homogeneous data formats. Nodes may vary in their memory size, processor counts and their speed, IO capabilities, and so on.

Nodes in a multicomputer basically trust one another, since they are all collaborating to provide a common resource. (Individual applications operating on the nodes of the multicomputer still need to be protected from one another's actions: the analogy aimed for is that of a timesharing system.)

The intended application domain for the work described here is moderately coarse-grained parallel computations—for example, a distributed database for online transaction processing (*OLTP*). In this environment, multicomputers offer the potential for smooth incremental scalability in performance and capacity—changes that require a processor replacement in a uniprocessor system. Compared with tightly-coupled shared-memory systems, multicomputers offer a wider range of cost-effective scalability with greater fault tolerance.

By comparison with uniform-access-cost shared memory systems (or UMA), the multicomputer model is easier to scale up to large numbers of nodes (only the explicit inter-node traffic needs to be supported, not every single memory access). So far, UMA machines have scaled successfully only up to a few tens of relatively slow processors (e.g., the Sequent machines, which use the Intel 80x86 family), and less than about eight with faster processors (e.g., IBM 3090s, Crays).

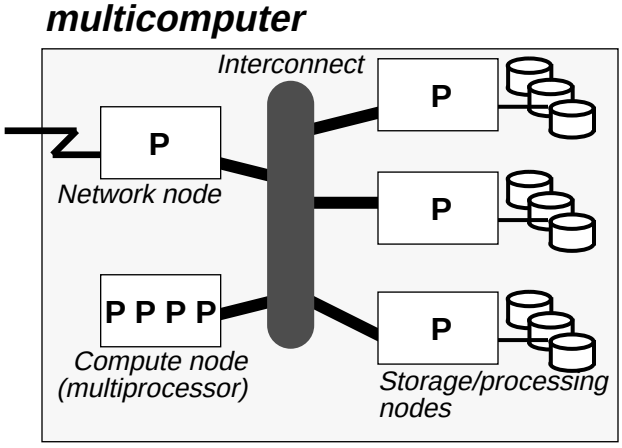

Figure 1: multicomputer system components (**P** = processor).

Compared against non-uniform-access-cost shared memory systems (NUMA), the additional isolation between the processor nodes in a multicomputer makes it easier to provide fault-tolerance, since the interconnect acts to isolate failures and limit the spread of damage from a faulty process or processor.

Because the internal multicomputer interconnect is enclosed in a cabinet, along with the other multicomputer components, it is not subject to replay or masquerade attacks, so there is no need for encrypting data traffic on it. Moreover, it is reasonable to expect exceedingly low error and failure rates because of the controlled environment in which it operates. Finally, its small scale translates into greater freedom in choosing a topology that can provide good scalability, as well as sufficient redundancy so that partitioning cannot occur.

None of these properties are valid for LAN-connected multiprocessors, and their performance is much lower as a result.

1.2 Multicomputer interconnect

The performance of a multicomputer interconnect fabric is obviously a major determining factor in the performance of the whole machine, and in how far this can be scaled up. Table 1 presents some performance figures for recent interconnect types used in various kinds of non-shared-memory multiprocessor. It shows small-message round-trip latencies and large-message bandwidths for a number of interconnect / platform combinations. As the table suggests, the trend is clearly towards higher bandwidths and—at least as importantly—lower latencies for small messages.

Unfortunately, bandwidths are going up faster than latencies are going down—due in large part to the relative lack of attention that has been paid to the interface between processors and interconnects. With processor speeds climbing steadily, the penalty in application-instruction opportunities of increased latencies are getting worse. Latency is the main enemy, not bandwidth.

1.2.1 Delivery order

To ensure that the proposed solution covers the widest range of interconnect characteristics, the proposal here is based on a deliberately weak set of assumptions about the properties of the interconnect as far as its delivery guarantees and failure modes go.

The multicomputer interconnect fabric is constructed from a set of packet switches, with individual packet-level routing. Because each packet is routed individually, the packets from a single message can be delivered out of order at the recipient. Worse, because there is no low-level acknowledgment on the delivery of a message at the

Table 1: some multicomputer interconnect performances expressed as round-trip latencies and bandwidths to and from a node.

System	*Interconnect*	*Latency (us)*	*BW (MB/s)*
HP-DUX/68030	Ethernet	~300	1
VAXcluster (780s)	CI	1000	8
HP 9000/700 "Snake farm"	FDDI (Medusa card)	~350	12
DECstation 5000	Fore Systems ATM card	~70	4
Tandem cluster	internal bus	?	2×20
HP Mayfly	hexagonal mesh	30	48
Intel Paragon (Touchstone)	rectangular mesh	76	25
Inmos T800	T8 link	100	1.6
Inmos T9000	T9 link+C104	~10	10
PageServer[a]	100m fiber	10	125

a. Paper design only.

interconnect level, entire messages can be delivered out of order.

The interconnect will never replicate data of its own accord.

The interconnect may make available a circuit-switched mode for transferring large amounts of data.[1] Because a circuit-switched connection will always deliver packets in order, the difficulties associated with out-of-order arrival at the recipient are removed. However, it introduces a complication at the sending node's interface, since true-in-order sending would prevent a node from transmitting any packets if the virtual circuit locked up for some reason—e.g., if an attempt was made to establish a connection to a dead node. It is important that the sending node be able to continue work during the potentially long timeout necessary to detect such a case, and this will require some form of non-FIFO packet send ordering.

1.2.2 Interconnect failures

Individual components of the interconnect fabric—switches, links, and connections to their nodes—can fail independently, but there will be sufficient redundancy in the interconnect fabric to prevent a partition occurring: that is, no two parts of the fabric can continue operating independently after a failure.

The interconnect fabric must not deadlock or livelock if one or more of the system components (nodes, switches or links) fails—to do this, it must have a mechanism by which packets destined for a broken or unreachable node are eventually garbage-collected. (This could mean deleting such a packet, or sending it to a "dead-letter office" for examination.)

Failures (and subsequent repairs) will be very rare events. This means that managing failures and recovery at upper levels of the software will be adequate from the point of view of performance, and no low-level error recovery or retries will be necessary. In particular, individual messages may be lost or discarded, if they are sent to a failed node, or through a broken or unreachable switch. However, this happens only during those (rare) intervals during which an interconnect or node failure has occurred but has not yet been detected.

The corrected bit error rate of the interconnect will be exceedingly low: so low that software checksums over messages are not required. This suggests that error rates of less than 1 in 10^{20} bits transmitted are probably required: the model that software should assume is that the interconnect is as reliable as a computer backplane, except in those very rare, few moments between occurrence and detection of a hardware error, during which messages may get discarded as a result of being mangled in a detectable way.

The interconnect fabric may be able to provide information on congestion and failures. Obviously this will be put to use if it is available, but it shouldn't be necessary for correct operation.

1.3 Node failures

Individual nodes fail independently: there is sufficient redundancy in power supplies and cabling to prevent system failure as a result of a single component failure.

When a node fails, it can do so in a clean manner (fail-stop: the node stops sending messages as soon as a fault occurs—no bad messages are ever sent out), or a messy one, with no constraints at all on the messages sent out after the fault occurs before it is detected and the node shuts itself down.

Most existing work on fault-tolerance has assumed the fail stop model because it is so much more tractable, and because truly unregulated behavior seems to be overly pessimistic for all but the most rigorous, life-critical applications. In particular, the costs of addressing the arbitrary failure mode case, or of providing true fault-stop processors, are exorbitant [Cristian91, Perry86, Schneider84, Gray88b].

However, a gradually-deteriorating node is unlikely to exhibit truly fail-stop behavior, so the model that will be used here is of *fail-fast nodes*. In this model, a failing node emits a sequence of good messages, followed by a bounded (possibly empty) sequence of bad ones. The bound can be a number of messages or an elapsed time. In either case, it can be used to determine how often to perform internal self-tests.

The resulting design constraint is a need to protect against a few bad messages from a failing node rather than a long-lived malicious attack. The expectation is that the interconnect hardware will catch (and discard) completely garbled messages, but that software (or interface) may be able to emit a few messages that look valid at the per-packet hardware checksum level, but are in fact meaningless. The fail-fast model being used means that we don't have to defend against arbitrarily-prolonged attack under such circumstances, which reduces the strength of the protection measures needed.

1. Notice that the boundary between large and small is not necessarily the same as that used by the OS to distinguish bulk data transfers.

1.4 Expansion and repairs

An attractive feature of the multicomputer model is the support it provides for smooth, incremental growth. For this to be possible it is important to be able to add individual nodes and the portion of the interconnect fabric that supports them, as well as to repair and replace failed components—while the system is online. For example, the requirements for certain database systems used in telecommunication applications include minute-per-year total downtime sustained over a 30-year mission life. This can only be achieved by adding and replacing hardware without shutting the system down. (This has significant implications on the kind of connectors and physical packaging, as well as on the software.)

One approach would be to bind a switching element to each node, and add and remove these as a pair. (There may have to be a passive backplane-like component into which the nodes are plugged; each backplane unit will support a few such nodes.) Another approach is to provide complete, redundant switching fabrics, so that one can be taken down for repair or expansion independently of the other.

2 Message traffic in DMPPs

There are three different classes of message traffic important to efficient operation of a multicomputer. Together, these represent the operations that a multicomputer interconnect needs to support well. (Particular applications outside the scope of OLTP, such as very fine-grain parallel computations, may require additional support that is not addressed here.)

The traffic types are as follows:

- atomic operations
- short requests
- bulk data transfer

These are elaborated on below, together with some remarks on desirable semantics for hardware-assisted multicast.

First, here is some terminology that will prove useful later on:

- *sender*: a node that is transmitting a message
- *receiver*: a node that a message is being sent to
- *originator*: a software entity (e.g., OS, user-level application, or subsystem) that is sending a message—necessarily from a sending node
- *target*: a software entity that a message is being sent to—necessarily on a receiver node.

2.1 Atomic operations

The atomic single-word memory operations available in a shared-memory multiprocessor are very convenient for some applications. Such operations (if used carefully) are ideal for low-latency concurrency control [Anderson89, Anderson90h], and for low cost communication between cooperating processes. It would be nice if these could be extended to the multicomputer environment.

Consider the case of lock management in a distributed database. Most accesses do not conflict, so claiming a lock is only a couple of memory operations. In the shared memory case, this is easily achieved. In a software implementation of message-passing, even the best implementations take hundreds of microseconds [Spector82].

The cost is so high because performing a remote lock operation in software involves a message send, and then (at the receiver) an interrupt and two context switches and the reply processing, followed by a message reply. With hardware support, this could all be reduced to a couple of round-trip interconnect times plus the execution time of an atomic memory operation at the remote node—perhaps 5µs (for small values of 5), almost all of which would be interconnect transit time.

Such high-performance atomicity operations would be of considerable value in allowing control-intensive operations (e.g., lock management) to scale to large numbers of processors.

2.2 Short requests

Atomic operations are fine for single-word updates, but are not particularly appropriate where the goal is to send a multi-word message to a remote node. Instead of incurring a round-trip interconnect delay for each word, it would be faster to pipeline the transfers. This is typically done by sending one or more *packets* of data through the interconnect, each several words in length.

Packet sizes vary widely: from a few bytes (48+5 bytes for ATM), through tens or low hundreds (e.g., 128 bytes for Mayfly [Davis89]), up to thousands (FDDI) or even millions (HIPPI) of bytes.

The real goal is to transmit *messages*: arbitrarily-sized units of data. If a message is larger than a packet, dividing it up into packets and reassembling them at the receiver can be done by software (the traditional approach for LANs) or with hardware assistance. The costs of software assembly / disassembly can be large, and much effort has been expended in the OS and communications communities to find ways to streamline these processes.

It is convenient to divide the use of messages into two classes: *short requests* and *bulk data transfer*. This is consistent with various measurements of existing network traffic on both wide-area and closely-coupled systems, which show strongly bimodal distributions: most messages are small, but most bytes are shipped in large messages [Bershad90, Caceres91, Cheriton87a, Mogul91, Pawlita81].

The prototypical interaction for short requests is client-server remote procedure call (RPC) [Nelson81, Birrell84]. RPC is usually dominated by small argument and result lists; the amount of processing required is often quite small, so efficient transport and delivery is very important; high priority transmission and handling is often useful, since RPC requests may carry urgent information about the state of a computation; and the overheads of memory management and multiplexing in the receiver are significant factors in the execution cost of such requests [Schroeder90, Scott87a, Thekkath91, vanRenesse89].

RPC, which is by definition synchronous, can be generalized to allow asynchrony at the client and server [Lampson82]. This can support a wide variety of interaction models [Cohrs88, Otway87a, Gammage87, Gifford88,

LeBlanc83, LeBlanc84a, Lin85, Liskov87, Liskov87b, Liskov91b, Stamos90].

All these uses have similar message delivery requirements from the interconnect: low overheads at the sender and receiver, and low interconnect latency.

2.3 Bulk data transfer

It will often be necessary to move large amounts of data (up to several megabytes at a time) between the nodes of the multicomputer. Examples of this include inter-node paging, access to non-local storage devices (e.g., disks, tertiary storage hierarchies), and distributed join operations on large tables in a relational database.

In a traditional interconnect, bulk data transfers are managed in just the same way as small messages: the components of a large transfer arrive piecemeal, and are put together by the receiver, and (most likely) copied into their final destination. This can be expensive: processor memory-copy speeds have not kept pace recently with improvements in their instruction execution rates [Ousterhout90].

One technique to avoid the copy is to use optimistic packet placement schemes: assume that the next packet to arrive is next one in the current bulk data transfer, and place it accordingly. Although this has been shown to work quite well on a LAN [Carter89], the cost is high when the prediction is wrong. It is also not clear how well the consecutive-packets assumption will apply to the multicomputer environment, where computations are likely to use much more communication than in a LAN.

A second kind of bulk data transfer is typified by a pipeline between a producer and consumer. Each element of the stream is processed as it arrives (depending on the semantics, this may be possible even if the elements arrive out of order). This stream-mode bulk data transfer combines the need for efficient bulk data movement with a tight integration of inter-processor scheduling. Such a model is supported in the bulk data transfer model of the Xerox Courier RPC system [Xerox84g].

In summary: copy avoidance is crucial for bulk data transfer, and it is useful to support a stream-based producer-consumer model in addition to simply handling large one-shot data transfers.

2.4 Multicast

There are some algorithms where the same data has to be sent to a subset $n>1$ of the N nodes in the multicomputer in a short period of time. Although such dissemination can be achieved in $\log n$ time through the use of a software-based multicast tree, the latencies involved at each step include delivery of a message, fielding an interrupt and the forwarding of the message to further sites, so the constant of proportionality is large.

In all cases, some mechanism for identifying the list of recipient nodes is required. techniques that have been used include:

- true broadcast (send to everybody);
- filter at the recipient (as in an Ethernet, where the recipient matches a multicast address against a set that it is willing to accept);
- partial sub-setting by address, followed by recipient-filtering;
- exact list of recipients.

There is an obvious trade-off between hardware complexity (e.g., for managing variable-sized lists of addresses) against execution-time costs such as discarding packets that incorrectly survive the filtering process, which must necessarily be conservative.

2.4.1 Reliability-insensitive applications

For some classes of applications there are calculations in which large-n multicast is valuable (e.g., communicating the result of a global sum to all the nodes for the next step in a relaxation calculation), even if it is not 100 percent reliable: if the data is not received at all the nodes, some form of slower out-of-band signalling mechanism can be used to indicate this fact, and the data resent to those nodes that did not receive the first time.

In such systems, even unreliable multicast may be worthwhile: if the likelihood of successful delivery is sufficiently high. The cost may still be lower than a sequence of individual message sends if only a few nodes need explicit retransmission of the lost message.

2.4.2 Reliability-sensitive applications

For a database OLTP application, however, the number of recipients in a multicast will commonly be fairly small (perhaps in the range 2–6, bearing in mind the need for replication to a backup for redundancy). For this application, hardware multicasting support *may* be an unnecessary complication.

In this environment it is essential that the hardware multicast provide a *reliable* service: either it delivers the message, or the sender gets told *in a timely fashion* that one or more recipients did not receive it. The sender needs to *know* that the recipients got the message before proceeding to the next step. Note that the typical problem is not lost packets, but receiver buffer overruns.

Few hardware-supported multicast systems meet this reliability test: usually they provide no way for the sender to tell whether the message was successfully delivered to the memories of all the receiver nodes. Without a reliable count of the number of successful deliveries being made available to the sender, it is likely that slower software-based techniques will have to be used anyway, thus abrogating any possible benefit.

One approach to providing the kind of reliability needed is to deliver multicast messages via a spanning tree embedded in the switching fabric (e.g., DEC's Autonet network [Schroeder90a, Rodeheffer91]). During the fan-out stage, the multicast value is propagated down the tree to the leaf nodes. Once leaves are reached, the spanning tree can be "rolled up", passing data about the success or failure of delivery at each lower level back towards the root. A simple messages-delivered count would suffice: when it reached the root, the number of successfully-delivered messages could be communicated to the sender.

2.5 Summary

The three kinds of messages (atomic operations for concurrency control, short requests, and bulk data transfer) have significantly different requirements.

Multicast is orthogonal to these, and could be useful for all three message types, provided it has support for indication of successful delivery of messages. Since it is likely to be most useful for the short-request case, a restriction in the interconnect that limits reliable broadcasts to single packets may still be acceptable.

3 Sender-directed interconnect

This section presents a new proposal for interfacing to the multicomputer interconnect. It is explicitly designed to support the kinds of traffic identified above.

The approach taken here can be described as having:

- remote Fetch-and-Op;
- message-send operations (possibly with multicast);
- sender-controlled message placement at the recipient;
- mapping tables to eliminate the need for global address-space management and to provide protection against rogue accesses;
- hardware-provided multiplexing at the sender for very low latency.

These are discussed in more detail below. Figure 2 is a diagrammatic overview of the interconnect model.

Some key assumptions of this model (by comparison to regular LAN networking, for example) are that the number of senders per node is relatively small, and that memory is cheap. The result means that it is cost effective to have each receiver node dedicate a page or two of memory to each sending node.

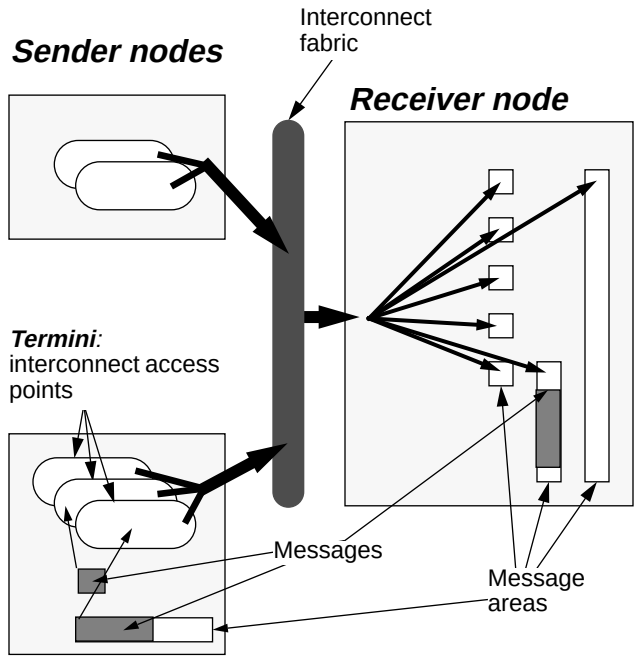

Figure 2: interconnect component overview.

3.1 Atomic operations

The model for atomic single-word operations is that of *remote Fetch-and-Op*: the operation is transmitted to the remote node, acted upon there, and the (possibly null) result returned once it is known.

Because the atomic operation is performed only once the request has been transmitted to the receiving node, it is a purely local request, and there is no need to hold a lock on the memory location for the duration of an interconnect round trip. Only once the result of the atomic operation has been determined, and the memory lock released, is the result returned to the originator.

Multiple outstanding Fetch-and-op requests may be in flight from a single sending node. There are no restrictions on where these are sent: they can all go to the same node, or even to the same memory address. Each request is treated as an individual atomic operation at the receiver. there is one minor complication: the potential for packet reordering in the interconnect means that the requests may not be executed in the order that they were issued at the originating node. If this is undesirable, additional software protocols will be needed to enforce a restriction to at most one outstanding Fetch-and-op per target (receiver, receiver slot, or address). Note that a combining interconnect is completely orthogonal to this arrangement.

3.2 Message send

The other main primitive is message-send.

All messages are sent to or from *message areas*—logically contiguous memory areas allocated at both sender and receiver. All the pages that are used in message areas are pinned in memory, and not allowed to be paged out. Although a message area is logically contiguous, it can span many physical pages that may not be physically contiguous. The length of a message area is not restricted to be integral multiples of a page, and nor is the start of a message area required to be on a page boundary. A message area is named by a tuple of the form <node,slot> at both the sender and receiver. There is no difference between "outgoing" and "incoming" message areas, other than through software convention.

Message areas are mapped to physical memory addresses by the interconnect hardware using a mapping table, described in more detail below. The local OS on each node is responsible for allocating message areas, and making sure that an originator is appropriately authorized to access them.

A message resides (or is delivered to) a range of addresses within a single message area. The address of the first byte of the message in its enclosing message area is referred to as its *offset*.

An originator sends messages from a source message area to a destination message area on a different node. It does so by identifying the source (its message area and offset), the target (another message area and a possibly different offset), and the message length. The key part of this is that

the sending node specifies where messages are to be placed at the receiver.

The outgoing data is written into the message area directly by the application: it doesn't need to be copied into an OS or networking buffer, and the OS doesn't need to be involved in validating virtual-to-physical address translations on each DMA message-send.

As data arrives at the receiver, it is placed into the chosen message area. This area can be mapped directly into the target process's address space. No copying need ever be done. (In the case of permanently-mapped areas on a non cache-coherent IO system, either the application will have to be cache smart, or the OS will need to purge the data cache after a message arrives.)

The use of the message areas at both senders and recipients means that applications can quote a form of "logical" address that is understood by the interconnect interface: they do not have to go through the local OS to have their virtual addresses converted into physical addresses, as is the case with other forms of IO. Furthermore, the message areas represent portions of memory that the OS has pre-approved for access directly by the application. No further checks are necessary at runtime (e.g., on every send- or receive-message call).

Multiple priorities (at least two) are provided, to allow urgent traffic to be sent out before less urgent data. (For example, so that high-priority control traffic can bypass a large bulk data transfer.)

3.3 Remote get

The message-send primitive described above is always sender-initiated. This means that doing a remote read (or get) requires that a message be delivered to the remote node, a context switch occur, and the required data be sent back. By this time, the requestor has also probably performed a context switch to a new activity, and so continuing the original processing will require a further one. Summary: two sends, two message deliveries and two context switches in the critical path.

The following *remote-get mechanism* would avoid one of these message deliveries and context switches (that at the receiver) by allowing the interface to perform the data retrieval directly. The mechanism works as follows:

- The sender transmits a special remote-get packet describing the data to be retrieved:
  - where it comes from (a message area, offset and length).
  - where to put it on the requesting node—again, in terms of a message area, offset and length.

  (The data-return protection key is the same as the one used to send the request to prevent one application masquerading as another.)
- The packet is sent to a slot at a remote node. When the packet arrives there, it is recognized as a special remote-get operation, and put it into a bin reserved for this kind of packet.[2]
- When all higher-priority outgoing traffic is completed, any remote-get operations at the next-lower priority level are executed. The data portion of the packet is interpreted as a send-work descriptor (to be described further below). Once the remote get is complete, the bin is marked as empty, and the next-highest-priority message will be sent—remote get or otherwise.

Since there is only one bin per priority level, a second one cannot be accepted until the processing of the previous one has completed. This means that the interface may have to refuse to accept *any* incoming packets until the current remote-get operation finishes. There are potentials for deadlocks here. Two steps are used to reduce the danger of this problem, although it cannot be completely eliminated.

The first is that remote-gets are executed with (logically) higher priority than sends initiated at the remote node. This has the advantage of emptying the remote-get bins quickly, but the associated potential disadvantage of denying service to the sender.

Second, the size of the data that can be retrieved by a remote get is restricted to bound the time that a bin can be locked up. To control this, the receiving interface has a register that contains the size of the largest allowed incoming remote-get operation. Any request that exceeds this value will be declared in error, and not processed. Since the primary intention of a remote-get assist in the interface hardware is reduced latency for small transfers, this should not be an excessive burden.

3.4 Notification and completion

Once a message (one or more packets) arrives, it may be necessary to indicate to the receiver that there is work to be done. The traditional way to do this is to deliver an interrupt on every incoming message (or packet!). This can obviously be expensive, and it is desirable to avoid it if possible. The goal is to minimize the number of interrupts while minimizing the response time to an individual request.

The approach taken here is to define a set of interesting events called *notifications*, and then to provide a mechanism by which only the minimum number of interrupts result from a stream of notifications.

Here is the set of notification events supported:

- *message-completed*: all of the packets in a message have arrived;
- *set-of-messages-completed*: a bundle of related messages (e.g., responses to a multicast) have arrived;
- *resource-exhaustion*: some resource (e.g., space for more notifications) has run out. (With a sender-initiated memory management scheme, this does not happen for data-space buffers.)
- *protocol error*: a Bad Thing has been detected.

Instead of causing a processor interrupt on each completion, the interconnect interface maintains a list of notifications—the *notification queue* in the processor's main memory. When a notification occurs, an entry is added to the notification queue identifying the incoming message and / or other cause. If the processor is still processing an earlier notification, nothing more need be done. Otherwise, if it has gone off to do something else, it is interrupted.

[2] There *needs* to be one bin per priority level. There *could be* many more—perhaps as many as one per slot in some implementations. This would require additional logic to chain the packets together in order of execution, however.

As a result, the number of interrupts decreases with increasing message traffic: under light load (when the disruption is lightest), most incoming messages will cause an interrupt. Under heavy load, when the need to suppress interrupts is greatest, it is quite likely that the processor is still processing a previous request, and does not need to be interrupted: it will find the new notification as soon as it has processed the earlier ones.

3.5 Mapping tables

Messages are sent from and delivered into message areas. These are just simple linear address spaces that occupy one or more physical pages in a node. The interfaces need to convert addresses in message spaces into physical addresses. They do this through per-node tables called *mapping tables* (Figure 3).

A mapping table is an array of tuples indexed by a *slot number* (a smallish integer). Each tuple describes one message area, although multiple tuples can describe the same message area, if needed. The tuple provides three distinct functions:

- an indirection between message addresses and physical ones;
- a protection mechanism to prevent malicious or accidental contamination of one message area by another;
- a (pointer to a) place where notification data can be stored.

The fields required to represent these functions are described here in turn. They are diagrammed in Figure 4.

3.5.1 Address mappings

Each slot describes a message area. The following fields are used for this:

- byte length (32 bits);
- page-map pointer (32–64 bits): the physical address of a vector of physical page addresses (the *page-map*); one entry for each page that this message area spans;
- start offset (32 bits) of the first byte in the message area from the start of the first page in the page-list.

The page-map entries are established at the time the physical pages are allocated to the message area and pinned into memory.

The page-map vector is outside the tuple so that it can grow to an arbitrary length, and so that it can be shared by multiple slots. The start-offset field allows a message area to be smaller than a page, and multiple message areas to share a single physical page.

The indirection provided by the page-map simplifies the management of the logical to physical address mapping. In particular, the interconnect interface doesn't have to be able to read the processor's TLB, nor does it need to have the mapping loaded on demand (e.g., by generating an interrupt for a miss). One or other of these would be needed if packets contain addresses in the receiver's virtual address space. At the same time, the page map at the receiver relieves the sender of knowing the virtual to physical translations in use at the receiver—indeed, the sender does not even have to knbow the receiver's physical page frame size.

If desired and available, the page-map can be set up to map some or all of a message area onto non-volatile memory. This may be useful for avoiding copies in nodes that manage response-time sensitive traffic that must be made non-volatile before a response can be given (e.g., responses to a multicast prepare-to-commit request).

The page-map is used as follows. Suppose an incoming packet has been validated and should now be written to the receiver's physical memory. Its logical start offset into the message area is contained in the packet itself—suppose this is address S_p. If the message space start offset is S_m, the index in the page-map of the relevant physical page is given by $N_p = (S_p+S_m)/\text{page\_size}$. The packet is written starting at offset $(S_p+S_m) \bmod \text{page\_size}$ within the page pointed to by this slot in the page-map. At each subsequent physical page boundary crossing, the next physical page address is retrieved from the page-map.

3.5.2 Protection

The goal of the protection mechanism is to provide a simple, lightweight scheme to protect against rare, accidentally-misdirected messages. As previously described, it is not necessary or intended that it be capable of withstanding a determined assault by a malicious adversary.

Protection is provided by requiring that each incoming packet have the same protection key as the slot. Keys are moderately sparse (32 bits), since this represents a

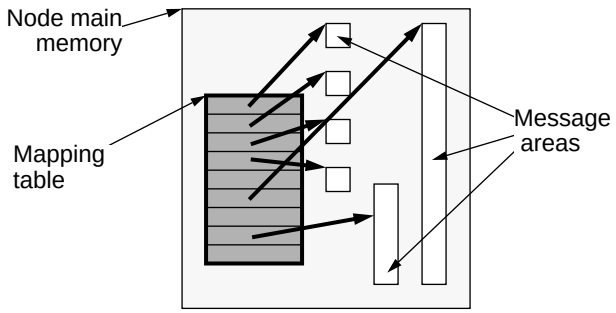

Figure 3: mapping table and message areas

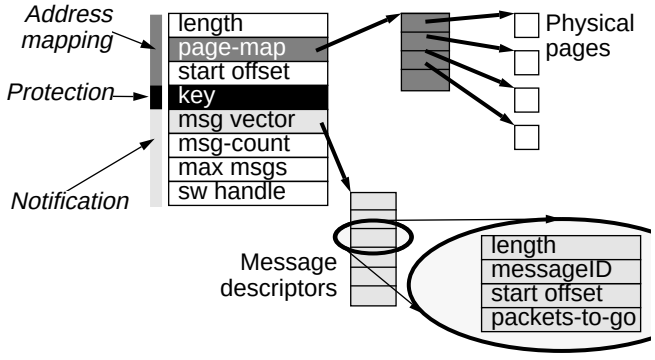

Figure 4: contents of a mapping table slot

reasonable compromise between ability to detect accidents cheaply and the cost of storing and transmitting the keys.

Packets are assigned their keys by the sending interface. To prevent an untrusted originator process "faking" keys, the key value can be set only by the trusted OS. Because of this, they can be relatively small integers.

> An alternative scheme would be to have capability-like keys that could be wielded by application-level code. This scheme has two main disadvantages: (1) the keys would need to be much sparser to avoid malicious attempts at application level to subvert security; (2) distribution of the keys used for bootstrapping would become very hard, since they would have to be transmitted by some out-of-band mechanism.

The message area in each slot is marked read-only or read-write for both incoming and outgoing traffic. This could be done by designating one of the bits in the key field a "write-enable" bit.

3.5.3 Notification

The main problem encountered with notification is not the notification mechanism itself, but the difficulty of determining when a message has arrived given that packets—including those intended for a subsequent message to the same message area—can arrive in any order. Determining completions—all the packets for one message have arrived—is the main challenge.

Consider first a restriction to have just one message at a time being in flight to a given slot. The notification data in each slot consists of a packets-to-go counter and a messageID field. As each packet arrives it decrements the counter; once the count reaches zero, a notification occurs.

The arrival of a new message is detected by the packets-to-go count already being zero, and a new packet arriving (which will always have a non-zero packets-in-message count). It causes the packet counter (and the messageID) to be reloaded: the packets-to-go counter is set to 1 less than the number of packets in the new message (each packet contains this value).

> Note: all such counting schemes rely on the interconnect not duplicating packets, and the sender not retransmitting a message without the receiver first having reset the packets-to-go count.

The messageID field is used to avoid errors during message re-sends. If a packet from the first attempt has simply spent a very long time in the interconnect and eventually comes out of the woodwork just as the new message is arriving, it must not be allowed to decrement the packets-to-go counter. To make sure that this situation is detected, a messageID is included in every message. This is compared against the value stored in the receiver's slot: only if they match is the packets-to-go counter decremented. The messageID slot is reloaded at the start of a new message.[3]

Now, let us generalize this to allow multiple messages to be in flight to the same slot. This is done by keeping a vector of message descriptors: one for each outstanding message that is allowed to be in flight at a time. This allows delivery of a notification only after multiple messages have arrived at a slot, which will be of use in certain classes of multicast-reply protocols.

The per-slot data structure consists of a (pointer to a) vector of message descriptors, indexed by the low-order bits of the messageID transmitted in every packet. When a packet arrives, its messageID is extracted and the appropriate message descriptor determined. This descriptor contains the full messageID (to detect long-lost packets that suddenly appear) and the packets-to-go counter for this message. Once the message descriptor has been identified, the two fields are treated just as described above.

In addition, there is a messages-to-go counter in the slot. The receiver uses this to declare how many messages must arrive before it is given a notification. The default value (zero) means "notify on every incoming message." If it is non-zero (only non-negative values are allowed), a completion merely decrements the counter. Only when it reaches zero is the notification delivered. A typical use for this counter would be to have notification delayed until all the (fixed-size) responses to a multicast had returned.

3.5.4 Multiplexed receive areas

One of the disadvantages of the "plain" notification scheme described here is that it is not enormously efficient for multiple senders trying to communicate variable-sized results back to the same place. The example I will use here is the multiple responses to a multicast. Some will be simple acknowledgments; others will be arbitrary amounts of data. This section discusses several potential solutions to this issue.

1. A purely sender-based scheme could achieve the right effect by using a different slot for each response. The disadvantage would be the number of notifications generated: one per message, rather than one to indicate that all the responses were in. (This is because we do not have cross-slot counting mechanisms.)
2. If the senders trust one another, the equivalent effect can be achieved by a form of receiver-controlled memory management. In the initial request (or by some convention previously agreed), the node sending out the multicast assigns areas fixed-size chunks of memory inside a single multicast-receive slot. Each multicast recipient is given a different chunk; when they reply, they put the fixed-length portion of their reply here. If they have a longer variable-length portion, they first send that to a separate slot, and include a pointer to it in their fixed-length response. This corks much better: the messages-per-slot counter can be used to delay a notification until all the fixed-length responses are in.

   Note: with our current out-of-order message delivery, there is the (slightly irksome!) possibility that the variable-length portion has not yet arrived, even though the fixed-length portion is in.

3.6 Protocol errors

There are various things that can go wrong—and be detected as having gone wrong—with this interface. Such protocol errors (e.g., sending a message to a message area that is too small to receive it all) will generate an error

[3] Another term for this is *incarnation number*—a value that is changed each time a new message is started.

notification on the interface that detects them, and the operation will be aborted.

For example, when a packet arrives at the interface at its receiving node, a set of validity checks are performed. Violation of any check will cause an error notification to be delivered to the receiving node.

1. The slot number in the packet is compared against the length of the mapping table.
2. The keys in the packet and the slot are compared to see if they match.
3. The start and length of the packet is compared against the size of the message area.

If the error is detected at the receiver, the receiver OS will then be given a chance to determine whether the sending node should be shut down or otherwise reprimanded. This is much easier to implement than telling the sender that it has committed an error. It may also help to slow down the sender (which is presumably awaiting some form of response) and so give the rest of the system more time to disable and reboot the faulty node.

3.7 Lost packets

What happens in either of these schemes when a packet is lost in the network for some reason? The problem is that the packet-count will be stuck at some non-zero value (probably 1), and never get decremented to zero: the missing packets simply don't arrive. Letting the system hang in this state is obviously undesirable.

One approach would be to have the receiver expect "heartbeat" messages if there is nothing else to send, and clean up the mess after a timeout if one of these doesn't arrive. Advantages: none. Disadvantages: receiver-initiated; the time-outs are likely to be quite long.

An alternative approach is to preserve the sender-manager nature of the protocol, and do one of the following:

1. Have the sender timeout and send a separate message to a different place in the receiver's slot saying "clean up the mess, please". Advantage: sender-initiated. Requires no extra hardware support. Disadvantage: there may be no extra space, and this scheme is also vulnerable to repeat failures, which could cascade and use up all the space.
2. Have the sender timeout and send a special one-packet message, flagged so that it would generate an error notification. (It would reuse the same messageID as the original message, be sent to overwrite the start of the original message, and bypass the packets-to-go counter.). Advantage: sender-initiated; disadvantage: timeout is expected length of service invocation.
3. As in the previous case, but have the hardware generate an ACK packet back to the sender on receipt of a complete message. Failure to get an ACK would then be grounds for proceeding as above. Advantage: faster time-outs (expected message delivery time, not service time), may be necessary for in-sequence message delivery; disadvantage: requires extra message in normal case.

I suggest that the second of these approaches is the best.

4 Paying the piper

The protocols that will be used on top of the Hamlyn interface are the subject of a separate document. A very brief outline of some of the issues that need to be addressed is provided here.

The sender-managed memory management will streamline a great many protocol issues—in particular, it will eliminate the need for a data copy. (This is particularly important for a storage server, where the data will typically not be looked at, but merely passed along to its ultimate consumer.)

The largest issue in protocol design is that of memory management of the message areas. The sender needs feedback from the receiver to know if the memory it has sent to can be reused. The obvious approach to use here is some kind of sliding-window protocol. Note that the "this is free ACK" can (and should) be piggybacked on top of the normal response to a synchronous RPC—i.e. the cost should be near zero, since such responses are needed for other purposes as well. (If desired, the same ACK can be used to indicate that a saved copy of the message being sent can be discarded, although the failure assumptions used here are probably such that recovery is needed so rarely that it can more cost-effectively be provided at a higher level.)

For short-message protocols like RPC, the expectation is that an approach like that of Bershad's LRPC [Bershad90] will have constructed the argument vector for the common (intra-node) case in a way that makes it trivial to convert it into a short message. This is most easily done by building the argument vector directly in a message area, leaving enough slack at its head for the inter-node RPC header to be added.

A technique known as active messages [vonEicken92] has pioneered the technique of embedding the addresses of software protocol handlers in the messages themselves: a couple of simple base / limit checks, and the callee can be off and executing code related to exactly the kind of message in hand in no time!

By mapping the message areas for such responses directly into non-volatile RAM, we can bypass an additional copy step needed to achieve fault tolerance support.

4.1 Bulk data transfer

The model of sender-managed memory management espoused above requires that there be physical memory pinned down at the receiver for each valid sender. This is reasonable for the memory needed for small requests (because it will only be a few pages), but is not sensible for large data transfers, where space would have to be pre-allocated for the largest possible transfer from each sender.

Instead, "large" transfers like this are managed by pre-negotiating space immediately before sending. (The meaning of "large" is defined by the receiver since it is the one that has to make the memory allocation decisions.) A large send might consist of the following steps:

1. *sender*: request space to send into;
2. *receiver*: allocates space; fills out a new slot in mapping table; returns slot number and key;
3. *sender*: performs transfer;

4. *receiver*: processes data and then discards mapping slot and buffer once it is done.

The first step can be omitted if the receiver is initiating the transfer: it can supply the mapping table information at the same time that it requests the data.

The last step can be removed if the sender and receiver expect to be performing some more complex buffer management protocol than a one-shot send. Indeed, arbitrarily complicated buffer management protocols can be devised. For example, the sender and receiver may be operating in a streaming producer/consumer mode, in which case they may choose to use the allocated space as a circular buffer, and not discard it until the entire connection is finished with.

4.2 Software multiplexing

Where performance is critical (as assumed for much of this discussion), application-level protocols with direct access to the interface and message areas will give the best results. Where resource conservation is more important than raw speed, traditional protected multiplexing mechanisms can be used to provide multiple virtual channels across a single OS-provided node-to-node connection. This looks a lot like regular message passing, with the difference that the sender still has some control over how the memory allocation is done at the receiver: in particular, there is no need to do a copy at the receiver if all the recipients of a multiplexed channel trust one another.

4.3 Bootstrapping

At start-up time each node allocates a single page of memory to each other node in the system, and initializes the first *N* slots in the mapping table to point to these pages. This way, each node is ready to receive messages from every other node, and further communication across these channels can establish more specialized connections.

The keys chosen for the bootstrap slots will be a function of the sending node address the slot is being set up for, rather than just the node number itself. (The function will be well-known, but require sufficient effort to compute to limit the likelihood of a bad node accidentally trampling on data it should have no access to.) The first action of a new node should be to negotiate a change to these keys to ones that are derived in a less predictable way.

Dynamic addition of nodes to a running system can be handled in the same way as the regular bootstrap: a node cannot tell the difference between a node that isn't communicating with it and one that has been unplugged (or never added). To use the bootstrap protocol described above, each node simply reserves a single-page message area and slot for all the nodes that might potentially exist in the system.

If the maximum number of nodes is large, but the number active is small, this can represent a waste of memory. To eliminate this problem, a few nodes can be designated as "boot servers", who are willing to tie down the memory required to communicate with any new node. The other nodes reserve slots only for the nodes known to be active (as determined by the boot servers). If a new node is plugged in, it searches the system for a boot server willing to listen to it, and announces its presence. The boot servers vet the new arrival, and communicate its existence to the other nodes, which then proceed to allocate a slot for OS-to-OS communication.

5 Termini

This section provides detailed descriptions of the data structures and control registers needed to drive the interface at both senders and receivers.

All Hamlyn communication is done through *termini*—end-points for communication. There is one receiving terminus, and one or more sending termini per node (Figure 5).

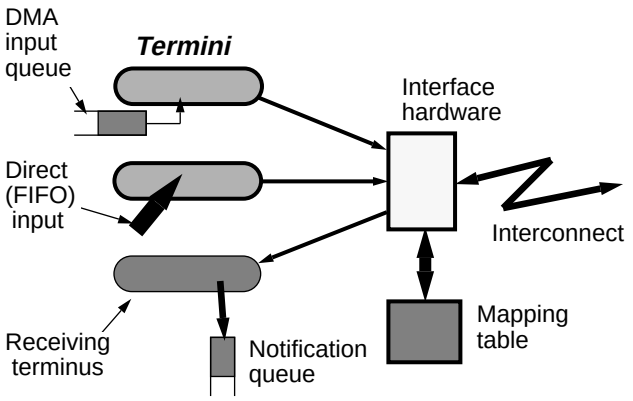

Figure 5: multiple termini

Termini are established by trusted OS software, which loads the privileged registers (those referring to physical addresses and the protection key). Other values are set by application-level software. All the termini share a single mapping table.

Each sending terminus has an associated priority. Our first prototype will have just two priorities: one for high-priority traffic (e.g., control messages), the other for low-priority messages (e.g., bulk data transfer). The priority scheme is simple: a higher-priority message will take precedence over a lower-priority one at the next packet boundary—i.e., it will preempt the lower-priority message send.

5.1 Establishing termini

The number of termini can vary dynamically up to a maximum set by the hardware. This number is intentionally not architected: this allows large systems, or ones where low-latency interconnects are particularly important, to use interfaces with more termini, while simpler ones can get away with only a few termini per node. (Once the hardware-provided termini have been handed out, software multiplexing—with its higher costs—will have to be used to provide further service.)

To establish a terminus in a PA-RISC system, the following need to be done:

- reset the hardware to clear its internal state (in case the terminus is being reused);

- set the soft physical addresses of the terminus's hardware registers (by writing a register in the interface's hard physical address space [James86]);
- load the protection key and the DMA input queue address and length (trusted OS code is required to do this, to avoid potential protection and / or security breaches);
- map the terminus into the originating application's address space, and make it readable and writable by the application's protection domain and privilege level.

5.2 The sender's interface

There is a single kind of sending terminus, but it can be used in two different ways: (1) to transfer messages from the processor's memory, (2) to perform very low-latency operations by accepting data directly from the processor's registers. The former interface style is called *DMA*, the latter *direct*.

The sender interface needs to support very low-overhead access for time-critical operations (especially Fetch-and-Op). It is also important that application programs can not be able to subvert the memory protection model of the sender OS. (Although they cannot damage memory by writing over it because of our send-only model, they could potentially transfer sensitive data to the outside if they were allowed to quote physical addresses to the hardware interface.) The use of message areas and the sender's mapping table provide this protection. This means that there can be a single sort of DMA terminus for use by both application-level code and the trusted OS and system components.

There are two kinds of hardware resource that need to be considered: the number of sending termini that the hardware can support (this is a function of the number of control register sets it is willing to handle), and the number of DMA engines that the interface contains. The latter can be smaller than the former if the hardware does software-transparent multiplexing behind the scenes. (In fact, this will be common: the standard configuration would probably multiplex a single DMA engine amongst multiple termini.)

5.2.1 DMA interface

With the DMA style of interface, the terminus is given a list of work items in main memory. Each work item represents a message send, so they are called, perhaps rather too predictably, *send-work-descriptors*. The contents of a work item are shown in Table 2.

Control of the terminus in its DMA mode is achieved by the set of registers described in Table 3.[4]

The list of work items is formatted as a circular vector in contiguous physical addresses (e.g., in a single page). The terminus operates by starting at the first work-item, processing it, and then proceeding on to the next one. When it reaches a work-descriptor with nodeID=0, it ceases work and awaits a prod from the processor before continuing. If the terminus ever reaches the end of the vector, it simply wraps around again to the first address in it.

Table 2: contents of a send-work-descriptor

field		*size (bits)*	*function*
receiver	nodeID	32	identifies receiver node
	message area	32	number of a slot in the receiver's mapping table
	offset	32	address of this message in the receiver message space
sender	message area	32	number of a slot in the sender's mapping table
	offset	32	address of this message in the sender message space
control	function	4+	**send** or **remote-get**
message	length	32	the number of bytes to send
	ID	32	used for notification
	sw_tag	32	for software protocol's use

The combination of the current position register and the active bit lets a processor to perform a race-free insertion of new work into the vector using the following protocol:

1. write the send-work-description, leaving the special marker set to "no work";
2. atomically write the special marker to indicate "work to do";
3. if the active bit is not set, and the current-position register shows that the interface stopped before executing the new work item, start it (write go into the control register).

The costs of sending a message are very low. In particular, there is no need for a protection domain switch into the OS for the message address and length to be checked and converted to a list of physical pages for the IO system—this has already been precomputed and stored in the mapping table.

5.2.2 Direct interface

The direct interface provides additional control registers in the interconnect interface. These registers are mapped into the address space of the application, and accessed directly by it with load/store operations. To send a message using the direct interface, the message destination and contents are written directly into the terminus, and then the send command is given. This mechanism is also the one used for Fetch and Op.

The direct-interface registers are listed in Table 3.

Doing a send operation involves the following steps:

1. wait until the sending terminus is idle;
2. write the receiver's node, slot, start address, and notification data into the control registers;
3. write the packet to send into the FIFO—this can either be a sequence of single-word stores into a single

[4] It may prove valuable to re-examine these in the light of the P1212 and P1212.1 IEEE standards in the area of interfaces and DMA control protocols.

Table 3: DMA control registers

register		*size (bits)*	*function*
DMA vector	start	64	Physical address of start of vector (writable only by OS).
	length	32	Byte-length of the DMA vector (writable only by the OS).
	current position	64	Physical address of the last send-work descriptor that the interface looked at. (This will be an empty work item if the active bit is clear.) Read-only by the application.
status	active	1	A Fetch-and-Op is in flight, or a send is still in progress.
	priority	1+	Priority level for this terminus (writable only by the OS).
	error data	??	Various forms of exception or error.
control	function	4+	Writing either **send** or **Op** here initiates the operation.
	protection key	32	Transmitted in every packet sent out by this terminus (writable only by the OS).

address, or a cache-line-flush to emit several bytes at once;

4. write send into the control register.

In all cases, if the values have not changed since the last operation, they need not be rewritten into the control registers.

A remote Fetch and Op is quite similar to a send, except that the operands are provided in place of a message body, and the result of the operation will usually need to be retrieved once it has completed:

1. wait until the terminus is idle;
2. write the receiver node, slot, address, and notification data into the control registers;
3. write the operands to the remote operation into the FIFO (this can either be a sequence of single-word stores into a single address, or a cache-line-write to a range of addresses);
4. write Op into the control register;
5. wait for the status register to read result available, and then retrieve the result from the remote operation.

A terminus can be active in both direct and DMA mode simultaneously. Direct-mode transfers are treated as if they were inserted at the head of the DMA queue.

The number of bytes sent in the message is the number that have been written into the FIFO. The size of the FIFO restricts the sizes of messages that can be sent by a direct terminus. As usual, a trade-off between cost and performance has to be made. The obvious size to pick is one large enough to contain the most common control messages: 256 bytes would certainly be adequate; 128 probably enough; 64 conceivably sufficient.

Table 4: registers in a direct terminus

register		*size (bits)*	*function*
receiver	nodeID	32	Identifies the receiver node.
	message area	32	Slot number in the receiver's mapping table.
	offset	32	Where to put this message in the receiver's message space.
message	msg_ID	32	for notification
	sw_tag	32	for software protocol's use
	FIFO for message data	*word or cache line*	FIFO into which data is put before being sent. Also used for the operands of the Fetch-and-Op.
status	active	1	A Fetch-and-Op is in flight, or a send is still in progress.
	result-available	1	A Fetch-and-Op has completed with a result.
	error data	??	Various forms of exception or error.
control	function	4+	Writing either "send" or "Op" here initiates the operation.
	result	64	Return value from the last Fetch-and-Op to complete.
	protection key	32–64?	Writable only by the OS; transmitted in every packet sent out by this terminus.

Letting an application supply its own protection key would permit it to masquerade as another, and this security violation is obviously not acceptable. This is why the key field is loadable only by the OS.

On the other hand, the name of the receiver message area can be set directly by an application. This maximizes the number of communication paths that can be supported from each terminus (they are relatively scarce resources). One cost of this is that the OS will have to perform system-wide management of protection keys if it is to prevent an application masquerading as another.

The terminus priority could perhaps be supplied by the application, but this would be at the cost of permitting some additional denial-of-service attacks, and for receivers to always maintain notification queues for all possible expressible priorities. It can be set only by the OS for this reason.

5.2.3 Synchronous operation

The basic mode of interaction with the terminus is asynchronous: the initiator fires off a request and (if needed) performs a rendezvous with a result some time later. For sends, the rendezvous may not need to be explicit.

For Fetch-and-Op, the model is more nearly synchronous: an operation is begun, and (pretty soon afterwards) the result will be needed before further work can proceed. The interface is asynchronous to allow for those cases where useful processing can occur in parallel with the remote

operation's invocation. However, the performance of Fetch-and-Op is such that its result will commonly arrive back in less time than a full context switch. When time comes for the rendezvous, this suggests that the initiator should spin and wait for the result, rather than do a context switch, thereby also incurring (usually bad) cache replacement effects.

Some results may take a long time to return if they experience congestion or a fault, such as sending to a node that has recently failed. Strategies that only spin and wait will pay dearly under such circumstances. A better approach is to employ so-called *competitive prediction strategies* for deciding how long to spin. The simplest scheme is to spin for a time that is as long as a context switch would take, and then yield the processor. Better is to take account of recent time-to-complete data: a discussion of various alternatives can be found in [Karlin91].

If multiple threads share a terminus, it is their responsibility to arrange for suitable mutual exclusion: the terminus provides no mechanism to achieve this.

5.2.4 Remote get

Remote get is a kind of synchronous operation: the sending terminus is blocked until the get returns. (This rule is here primarily to limit the rate at which such requests can be sent.) The send work descriptor is assembled as the body of the message, and the message is sent with an operation type of remote-get. When the message arrives at the receiver, it is put into a remote-get bin at the appropriate priority level (note: the requester's priority is used); the function field is forced to the value **send**, and the operation processed when there are no more higher-priority requests.

5.3 The receiver interface

The receiver interface is concerned almost solely with support of the notification system and its notification queue. This queue is managed in a very similar fashion to the one used to send DMA requests. This time it is the processor that needs to indicate to the interface (by writing a control register) how far it has progressed in handling the notification queue, and whether it has finished processing the current set of notifications.

When a notification occurs, the interface writes the data describing it into the next free slot in the notification queue, and interrupts the processor—if and only if the latter has indicated that it is finished processing the queue. It is important that the interface write out any cached data it may have regarding the slot on which the notification is declared before writing the notification into the queue.

The following items are written into a notification record:

- slot number (32 bits): which slot generated the notification
- messageID (32 bits): from the last packet
- message offset (32 bits): where this message starts in the message area
- message-length (32 bits): in bytes

As the processor deals with a notification, it clears the values to indicate to the interface that the entry is available for reuse. If the interface discovers that the place it is about to write a new notification is not cleared, it generates an interrupt to indicate that the notification queue has filled up. Unfortunately, it will also have to discard the data about which notification has just happened, which is really bad news. A possible solution here is to leave the offending notification data in control registers, and have the interface not accept any more incoming packets until the problem has been dealt with.

Each priority level has a separate notification queue.

5.4 The mapping table

Each interface has three control registers for the mapping table (the map is always allocated in contiguous *physical* space):

- map_start (64 bits): physical start address of map table
- map_length (32 bits) length of the table (in slots)
- cache_invalidate (32 bits): writing a slot number here discards any cached data about that slot

The receiver interface may cache data from the map in order to maximize its performance. (Whether or not a particular interface chooses to use a cache is an implementation choice. It must still support the cache management protocol described here.) This use of a per-slot cache provides greater flexibility than, say, a scheme where the map table has to fit into RAM on the interface.

To control the cache, a cache_invalidate register (16 bits) is provided on the interface. Writing a slot number here causes any cached state about the slot to be discarded (cf. purge). Writing a new value into either the map_start or map_length register invalidates the entire cache. The state of the cache is undefined at power-on (interface initialization) time.

5.5 Processor cache coherence

The standard DMA hardware cache coherence behavior will occur as messages arrive. For example, in current PA-RISC, nothing at all is done by the hardware: cache coherence is expected to be managed by the software. In this case, the sending processor will have to flush the caches of messages to be sent through a DMA channel, and the receiving processor purge the caches of data when messages arrive.

In other architectures, cache-coherence may be guaranteed by the IO interface, in which case this issue is not a problem.

6 Related work

The interface that most closely resembles that proposed here is that of the VAXcluster CI [Kronenberg86]. This uses an intelligent interface card to perform a variety of reliable and unreliable message-sends. Like Hamlyn, the sender can indicate where in the remote system's memory data is to be put. In the CI interface, the message area page-maps are pointers to VAX page table entries. The CI has a more complicated protocol between a local host and the interface, which includes send acknowledgments, and multiple priority queues sharing buffers, but there is no provision for direct application-level access to it. There is no protection mechanism, so the OS needs to intervene to set up all transfers. Finally, the CI is designed to work on a

network with a more restricted set of delivery properties (for example, there is no in-network routing). In fact, the only part of the interconnect that is not contained on the CI cards is the cabling and a passive star coupler. A similar approach to reducing interrupts is taken, but is based on the concept of send and receive requests being expressed in control packets that are moved between various lists as they make state transitions.

A scheme has been proposed for making IPC operations happen as a side-effect of memory-address references [Rosing92]. The control descriptors required to describe what needs to be done are stored on a per-address basis in an adjunct processor. Even if you like the model of programming through side-effects, there is no indication that the cost would be any lower than having the main processor perform the same work. The proposal does not address security, although it could be argued that the address-space protection mechanism could be used to protect particular side-effect addresses from unwanted application access.

A proposed design that was partially implemented for the VMP multiprocessor is described in a PhD thesis from Stanford [Kanakia91]. According to measurements made on a workstation and file server at Stanford, data copying, checksumming and encryption account for roughly 50% of the communication costs. To improve this, the thesis proposes off-loading protocol processing to an intelligent protocol engine that handles encryption, checksums, and header management. The disadvantages are that the controller is specific to a single protocol type (the design was couched in terms of VMTP), and its performance does not scale with that of the main cpu.

7 Conclusions

This paper used an analysis of the kinds of interconnect traffic in multicomputers to drive the design of a new sender-based interconnect model. The analysis suggests that there are three main kinds of operation required of a multicomputer interconnect: concurrency control, short messages for control traffic, and bulk data transfer. Hardware-supported multicast may be of benefit, provided that reliable knowledge about the success of delivery is available.

The bulk of the paper concerned itself with the description of a new interface to a multicomputer interconnect. The key concepts espoused are:

- an indirection table at the sender to map application addresses into physical addresses without intervention from the OS on each message-send;
- an indirection table at the receiver to map incoming data offsets into physical addresses to avoid the need to copy data;
- having the sender manage the memory allocation of the space into which it is placing data at the recipient;
- remote Fetch-and-Op as the system-wide synchronization mechanism;
- separation of notifications from completions, and support for multiple-message completion events
- the use of a DMA-controller-like interface between the interconnect and the host;
- hardware that maps multiple termini into user address spaces to support low-latency IPC operations direct from the application;
- a key-based protection scheme to check validity of messages at the receiver, and a mechanism by which every outgoing message is tagged with a (protected) key by the terminus;
- pre-negotiation of space for large transfers to avoid tying down memory for long periods of time.

7.1 Future work

This paper describes one manifestation of a Hamlyn interface. Many other implementations are possible: for example, a low-cost implementation could choose to move much of the processing on the sender side into OS software. Doubtless various different tradeoffs along these lines will be identified as this idea is explored further.

The next step is to develop a suite of IPC protocols and use them to improve and refine the interface. I hypothesize that the Hamlyn interface model will make it possible to achieve much higher performance, and such protocols will be significantly easier to write than for a regular LAN-like interface. Furthermore, I believe that it will be possible to develop highly efficient, streamlined application-to-application level protocols in addition to more traditional protocol suites such as the ISO / OSI and IP families.

There is also work to be done in modelling the performance of this scheme—both by comparison with more traditional approaches (as well as the van Jacobson *Witless* approach), and to enable finer-grain evaluation of some of the design choices and alternatives described here.

Finally, a prototype of the interface and an associated interconnect (based loosely on the Mayfly Post Office [Davis89]) will be used to put this design into practice. A preliminary step may be to use a rack of HP9000 / 700i cards for s prototype: these cards plug into a VME backplane, and can perform remote memory operations on each others RAM. This should make it easy to imitate (and experiment with variations on) the Hamlyn interface design.

7.2 Summary

The mechanisms proposed here allow both Fetch-and-Op and regular message traffic to be handled efficiently at both the sender and the receiver. They enforce protection through the use of a mapping table, which also provides an indirection between an address space available to the sender and that used by the receiver. The result allows local management of the virtual-to-physical address translation, and permits the interconnect hardware to operate without access to the processor's on-chip TLB.

Notification schemes allow for multiple message arrivals to be coalesced into the minimum number of interrupts. Extensions to the basic scheme also provide for low-overhead multicast receipt and remote memory retrieval operations.

I believe that the sender-managed interconnect approach is simple, elegant, and effective. By reducing software overheads, it will enable faster, more capable

multicomputers, and ones whose performance will more closely track growth in future processor speed than do current interconnect interfaces.

Acknowledgments

Marty Fouts provided the initial sounding-board for these ideas, and encouragement in developing them: they would not have formed without his willingness to listen. Al Davis and Robin Hodgson provided me with a great deal of insight into packet-switched interconnect fabrics, and made many helpful suggestions. Bob English has made some fruitful suggestions about how different implementations might be constructed. Mike Wenzel provided a very thorough analysis of an earlier draft. Their discussions and insights were invaluable in the process of putting this proposal together. Bill Worley provided encouragement during Hamlyn's early development, as well as technical feedback.

References

[Anderson90h] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, **1**(1):6–16, January 1990.

[Bershad90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, **8**(1):37–55, February 1990.

[Birrell84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, **2**(1):39–59, February 1984.

[Caceres91] Ramón Cáceres, Peter B. Danzig, Sugih Jamin, and Danny J. Mitzel. Characteristics of wide-area TCP/IP conversations. *Proceedings of SIGCOMM'91*, 1991.

[Carter89] John B. Carter and Willy Zwaenepoel. Optimistic implementation of bulk data transfer protocols. *1989 ACM SIGMETRICS and Performance '89 International Conference on Measurement and Modeling of Computer Systems* (Berkeley, CA). Published as *Performance Evaluation Review*, **17**(1):61–9, May 1989.

[Cheriton87a] David R. Cheriton and Carey L. Williamson. Network measurement of the VMTP request-response protocol in the V distributed system. Technical report STAN–CS–87–1145. Department of Computer Science, Stanford University, February 1987.

[Davis89] Al Davis. The Mayfly parallel processing system. Technical report HPL–SAL–89–22. Systems Architecture Laboratory, Hewlett-Packard Laboratories, December 1989.

[Gray88b] Jim Gray. A comparison of the Byzantine agreement problem and the transaction commit problem. Technical report 88.6. Tandem Computers, Cupertino, CA, May 1988.

[James86] David V. James, Stephen G. Burger, and Robert D. Odineal. Hewlett-Packard Precision Architecture: the input/output system. *Hewlett-Packard Journal*, **37**(8):23–30, August 1986.

[Kanakia91] Hemant Ratubhai Kanakia. *High-performance host interfacing for packet-switched networks*. PhD thesis, published as Technical report STAN–CS–91–1373. Department of Computer Science, Stanford University, July 1991.

[Karlin91] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):41–55, 13–16 October 1991.

[Kronenberg86] Nancy P. Kronenberg, Henry Levy, and William D. Strecker. VAXclusters: a closely-coupled distributed system. *ACM Transactions on Computer Systems*, **4**(2):130–46. *Presented at the Symposium on Operating Systems Principles, Orcas Island* (Dec. 1985), May 1986.

[Lampson82] Butler W. Lampson. Fast procedure calls. *Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Ca). Published as *ACM SIGARCH*, **10**(2):66–76, March 1982.

[Liskov91b] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, **9**(2):125–42, May 1991.

[Mogul91] Jeffrey C. Mogul. Network locality at the scale of processes. *Proceedings of SIGCOMM '91 Symp. on Communications Architectures and Protocols* (Zürich, Switzerland, 3–6 September 1991), pages 273–284.

[Nelson81] B. J. Nelson. *Remote procedure call*. PhD thesis, published as Technical report CMU–CS–81–119. Carnegie-Mellon University, Pittsburgh, PA, 1981.

[Ousterhout90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? *USENIX Summer Conference* (Anaheim, CA), pages 247–56, 11–15 June 1990.

[Pawlita81] Peter F. Pawlita. Traffic measurements in data networks, recent measurement results, and some implications. *IEEE Transactions on Communication*, **29**(4):525–35, April 1981.

[Rodeheffer91] Thomas L. Rodeheffer and Michael D. Schroeder. Automatic reconfiguration in Autonet. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):183–97, 13–16 October 1991.

[Rosing92] Matthew Rosing and James N. Thomas. Reducing message latency by making message passing transparent. *Proceedings of 25th International Conference on System Sciences* (Kauai, Hawaii), volume 1, pages 593–9, Veljko Milutinovic and Bruce Shriver, editors, 7–10 January 1992.

[Schneider84] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems*, **2**(2):145–54, May 1984.

[Schroeder90a] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. Technical report 59. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, April 1990.

[Schroeder90] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, **8**(1):1–17, February 1990.

[Scott87a] Michael L. Scott and Alan L. Cox. An empirical study of message-passing overhead. *Proceedings of 7th International Conference on Distributed Computing Systems* (Berlin, 21–25 September, 1987), pages 536–43, R. Popescu-

Zeletin, G. Le Lann, and K. H. Kim, editors. IEEE Computer Society Press, 1987.

[Spector82] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, **25**(4):246–59, April 1982.

[Stamos90] James W. Stamos and David K. Gifford. Implementing remote evaluation. *IEEE Transactions on Software Engineering*, **16**(7):710–22, July 1990.

[Thekkath91] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency RPC. Technical report 91–06–01. Department of Computer Science and Engineering, University of Washington, 1991.

[vanRenesse89] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. The performance of the Amoeba distributed operating system. *Software—Practice and Experience*, **19**(3):223–34, March 1989.

[vonEicken92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schuser. Active messages: a mechanism for integrated communication and computation. *Proceedings of 19th International Symposium on Computer Architecture* (Gold Coast, Australia), pages 256–66, 19–21 May 1992.

[Xerox84g] Xerox Corporation, Stamford, Connecticut 06904. *Courier: the remote procedure call protocol. Appendix F: bulk data transfer*, Xerox Systems Integration Standard 038112 Addendum 1a, April 1984.