An implementation of the Hamlyn sender-managed interface architecture

Greg Buzzard, David Jacobson, Milon Mackey,
Scott Marovich, and John Wilkes

Computer Systems Laboratory,
Hewlett-Packard Laboratories, Palo Alto, CA

As the latency and bandwidth of multicomputer interconnection fabrics improve, there is a growing need for an interface between them and host processors that does not hide these gains behind software overhead. The Hamlyn interface architecture does this. It uses sender-based memory management to eliminate receiver buffer overruns, provides applications with direct hardware access to minimize latency, supports adaptive routing networks to allow higher throughput, and offers full protection between applications so that it can be used in a general-purpose computing environment. To test these claims we built a prototype Hamlyn interface for a Myrinet network connected to a standard HP workstation and report here on its design and performance. Our interface delivers an application-to-application round trip time of 28µs for short messages and a one way time of 17.4µs + 32.6ns/byte (30.7MB/s) for longer ones, while requiring fewer CPU cycles than an aggressive implementation of Active Messages on the CM-5.

1 Introduction

Processors are rapidly getting faster, and message-passing multicomputer interconnections are doing the same, thanks to recent developments in Gb/s links and low-latency packet switches. But the cost of passing messages between applications also includes the overhead of crossing interfaces between the operating system (OS), a device driver, and the hardware, which can be orders of magnitude more than the cost of moving a message's bits across the wires.

Hamlyn is an architecture for processor-interconnection interfaces that addresses this difficulty. It achieves both low latency and high bandwidth, isolates applications from each other's mistakes, and supplies a rich set of message-delivery semantics. It does so by exploiting several techniques:

- *Sender-based memory management.* Senders, not receivers, choose the destination memory address at which messages are deposited. This means that messages are sent only when the sender knows that there is memory space for them, eliminating buffer overrun and retransmission under heavy loads.
- *Direct application access to interface hardware.* Send and receive operations require no OS intervention, yielding very low latencies.
- *Zero-copy protocols.* Data are transferred directly between application memory and the network with no memory-to-memory copying or page remapping.
- *Automatic message reassembly.* The interface allows out-of-order packet delivery in order to support adaptive routing networks, which have greater throughput and fault tolerance [Davis92].
- *Data movement and message arrival notification are separate*. Data can be moved without interrupting the remote host if desired, which provides greater application control and lower overheads [Thekkath94].

The original Hamlyn design [Wilkes92, Wilkes95], which incorporated most of these features, was intended to support a packet-based, fault tolerant, adaptive routing network for a large-scale, MIMD multicomputer, derived from the Mayfly project [Davis92]. This paper extends the original Hamlyn work by describing:

- performance data from a working prototype;
- improved methods of message arrival notification;
- a more powerful packet counting scheme that supports generalized group-receive semantics;
- layered protocols that provide in-order message streams and application buffer management.

We describe the Hamlyn architecture and our application interface library, presents performance measurements of our prototype, discusses related work, and then summarizes what we have learned.

2 The Hamlyn architecture

Hamlyn was intended to support scalable, concurrent, fault-tolerant applications, running on a MIMD multicomputer or a closely-coupled computer cluster. Such applications often require high-bandwidth bulk data transfers, low-latency control messages (a few microseconds per round-trip), and multiple, independent protection domains provided concurrently on each processor.

This paper appears in the proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96), Seattle, Washington, 28–31 October 1996. The USENIX Association has exclusive rights to publish the paper until October 1997. © Copyright Hewlett-Packard Company 1996. All other rights reserved.

For low latency, Hamlyn gives applications direct access to the interface hardware for sending messages with no OS intervention. It provides a fast, low-cost, message-arrival notification mechanism that does not require interrupts or system calls in order to receive messages. (Interrupts may be used as an optional alternative, in which case Hamlyn tries to coalesce them instead of delivering one for each packet.)

For high bandwidth, Hamlyn includes a scatter-gather direct memory access (DMA) capability that frees the host processor for other operations during long transfers. Applications can use it directly: it does not require OS intervention for each use. This allows Hamlyn to avoid all memory-to-memory copying in the host.

For security in a multiple-user environment, Hamlyn prevents mutually suspicious applications from sending, reading, or overwriting each other's data even though they use the interface hardware directly. This avoids the need to statically partition a multicomputer, or having to use gang-scheduling and interconnection fabric draining for inter-application protection, as on the *CM-5*.

Hamlyn deliberately exploits several features of the short-distance interconnection networks commonly used in modern, multicomputer systems:

- *Very low transient error rates.* Dedicated, enclosed, multicomputer interconnection fabrics are better thought of as extended backplanes than as unreliable networks—they rarely lose or corrupt packets. Hard failures can occur, but transient errors are so infrequent (perhaps one every few months) that it is reasonable to handle them using high-level, application-program mechanism, such as aborting and restarting a transaction [Saltzer84].

  This means that automatic retransmission by a lower level of a protocol stack is unnecessary, improving performance; that packet reception need not be acknowledged, eliminating a potential cause of deadlock; and that transmission buffers can be released as soon as their contents have been sent, simplifying buffer management.
- *Hardware flow control in the interconnection fabric.* This avoids packet loss by applying back pressure to senders when resources fill up.
- *Small packet sizes.* These permit simpler, faster switches and better throughput guarantees, at the cost of requiring message segmentation and reassembly in the interface.
- *A physically secure network.* In such networks, messages need not be encrypted to protect them against eavesdropping.

Hamlyn was designed with a RISC-like philosophy: to make common cases fast and less common ones possible.

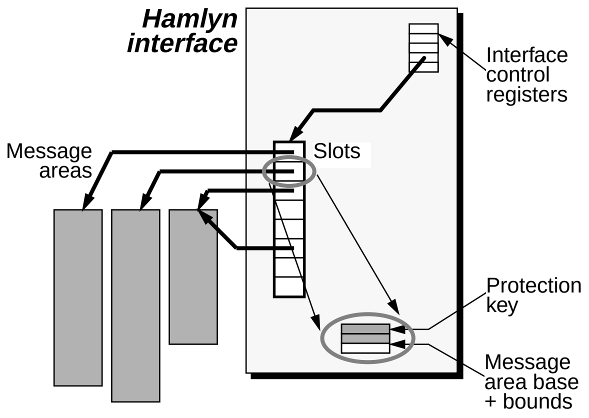

Figure 1: message areas and slots.

An explicit goal was that Hamlyn should be simple enough to be implementable using hardware state machines, since programmable controllers are often slow.

The next subsections describe the features of the Hamlyn design that allow these goals to be met.

2.1 Sender-based memory management

The first—and perhaps most important—feature is *sender-based memory management*, which is a technique to avoid software-induced packet loss. Packet loss is a serious problem in low-latency data communication systems because coping with it usually means retaining transmission buffers, acknowledging packet reception in order to release buffers when they are no longer needed, and using a low-level time-out mechanism to trigger retransmission. Moreover, the problem usually occurs under heavy loads, when retransmission will only make it worse, so even a low rate of packet loss can produce a much higher rate of message loss.

There are two main causes of packet loss: interconnection network problems, such as damaged or lost packets, and receiver buffer overrun. Our design assumptions legislate away the former, and we use sender-based memory management to prevent the latter.

The basic idea is to determine a message's final destination in a receiving host's memory before sending it, so that receiver buffer overrun is impossible. The receiving network interface places incoming packets directly into their final resting place instead of leaving them on an interface card or temporarily copying them elsewhere in the receiving host's memory.

2.2 Slots: naming and protecting message areas

Data are sent to and from *message areas*, which are contiguous regions of an application's virtual-memory address space, protected by OS mechanisms in the usual way (Figure 1). Message areas are *wired down* (pinned into memory) while they remain allocated. This is a deliberate design decision that trades greater physical memory use for lower latency and greatly simplified interface design.

Message areas are referred to by *slots*, which are in turn indexed by small integer *slot numbers*. A slot contains base and bound registers for the message area to which it refers (several slots may refer to the same message area), and a protection key that must also be held by applications wishing to send messages to it. A slot is implemented by a data structure that can only be modified by the OS; this data structure lives in the network interface hardware in our prototype.

Memory addresses in Hamlyn messages are represented by <slot-number, offset> tuples; this indirection allows senders to be isolated from the details of virtual and physical memory addressing at receivers. Since packets can potentially arrive out of order, each one needs to be self-describing. This is accomplished by having the Hamlyn interface add a header to each packet that it sends out, as shown in Table 1.

Table 1: packet header format (slightly simplified). Each line represents 32 bits.

Destination host ID	
Slot number	Metadata index
Protection key (64 bits)	
Packet offset	
Packet length	
Delta (used in packet counting)	
Metadata length	Flags
(user data follows here …)	

When a packet arrives at a receiving interface (Figure 2), the interface locates the named slot, adds the base address of the target message area to the offset specified in the header, and then moves the packet's data to that address using DMA, after checking that the packet will fit into the message area. When the last packet of a message arrives the interface will also notify the receiving application if desired (see section 2.6).

To ensure that data cannot be written into a buffer without permission, the receiving interface compares the protection key in the packet header to the one in the slot. Only if the keys match is writing allowed. Hamlyn protection keys are large (64 bits) and sparsely allocated to provide good inter-application protection.

Application software can often be simplified if a message carries a small amount of out-of-band *metadata* to be deposited in a separate buffer, so our prototype Hamlyn interface allows up to 60 bytes of it in the first packet sent. Secondary base and bound registers and a protection key may be used to used to prevent unauthorized overwrites of metadata at the receiver (see section 2.7).

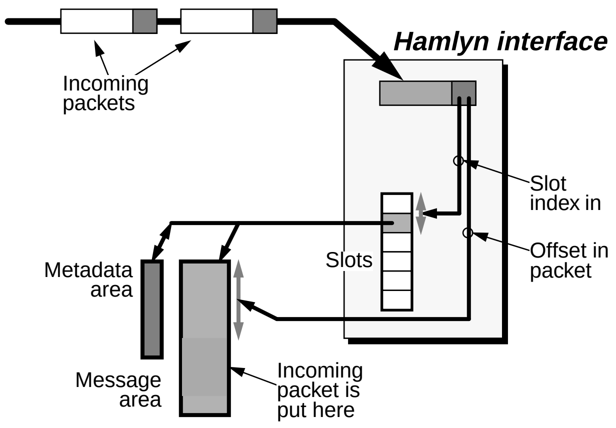

Figure 2: processing an incoming packet.

2.3 Sending termini

Hamlyn gives each sending application a private hardware *send terminus*, implemented as a set of control registers and a FIFO work queue in the interface card. These are mapped into the application's virtual-memory address space and protected by OS mechanisms in the usual way. In our prototype, each work queue holds up to 63 entries, so that applications can quickly post several messages without blocking. When the interface sends a message, it writes a sequence number in a prearranged, per-terminus, host memory word; this number, modulo the work queue's size, identifies the corresponding entry, thereby telling the application that the entry and any buffer memory associated with its message can be reused.

Short messages are pushed from a host processor to the terminus queue using ordinary STORE instructions. We call this *direct I/O*. To send a message, the application writes a transmission-request block—basically a packet header—into the send terminus' work queue, followed by any metadata, then the data. It then notifies the hardware of the new entry and proceeds to other work; no system call or interrupt occurs when sending a message.

Long messages are pulled from host memory by the interface using asynchronous DMA. Our prototype prevents applications from accessing data belonging to other applications in the following way. Each send terminus has 8 base and bounds registers, settable only by the OS, that are used to identify special *send buffer area*s that are named by small integer *send buffer tag*s. A single message can contain parts from one or more of these areas; the Hamlyn DMA engine interface gathers them up on the fly as the message is sent out.[1] To send a long

[1] Our original proposal [Wilkes92] used slots instead of special send message areas for this purpose.

message, the application writes a transmission-request block to the terminus' work queue that consists of the header followed by a sequence of <send_buffer_tag, offset, length> tuples describing the location of metadata and data. It then notifies the interface of the new entry.

The send terminus automatically segments messages larger than a single packet, replicating the header in each packet—except for the offset field, which is adjusted automatically to reflect the address of the new packet at the receiver. All metadata are put into the first packet.

The Hamlyn interface interleaves packets from all the send termini with non-empty work queues, providing approximately equal bandwidth to competing processes sending large amounts of data and, in the absence of network delays, bounding the time any message waits in a work queue. This scheme could be embellished with priorities, although our prototype didn't include them.

2.4 The metadata area

Message areas are intended to receive most incoming data, but we needed three other storage structures for each arriving message:

- a place in which to deposit metadata;
- packet counters for the message-arrival notification mechanism (see section 2.5);
- information for a finer-grained, per-message protection scheme (see section 2.7);

There might be hundreds of application processes, with hundreds of interleaved, concurrently arriving messages per process, so we might need tens of thousands of these structures—too many to store on the interface card.[2] We chose one mechanism to solve all of these problems: a 128-byte *metadata entry* is provided for each expected message (Table 2). These entries are arranged in a vector, called a *metadata area*, whose base and bound are stored in a slot data structure. It lives in a receiving application's virtual-memory address space. Each packet header specifies the index of a metadata entry associated with its destination slot, which may be thought of as a message identifier. A metadata entry may be reused as soon as all packets of a message referring to it have been assembled by the interface and processed by the receiving application. A slot must therefore have enough metadata entries associated with it to accommodate the largest number of messages that might arrive concurrently.

2.5 Packet counting

In some interconnection fabrics, packets of a segmented message may arrive at a receiving interface out of order, so a mechanism is needed to determine when all the

[2] The Myricom LANai 2.3 network controller IC that we first planned to use only supported 128K bytes of memory for all of the control program, slot and terminus data structures, and packet buffers.

Table 2: a metadata entry.

User metadata (60 bytes)
Packet accumulator
Paranoid-mode protection key
Paranoid-mode message area low limit
Paranoid-mode message area high limit
Hamlyn library receive class pointer

packets of a message have arrived and the receiving application can be notified. Hamlyn does this by maintaining a 32-bit *packet accumulator* in the metadata entry for each expected message. The value of the accumulator starts out at zero and returns to this when all the packets have arrived [Jacobson95].

Each send terminus has a 32-bit *packet counter*, which is initialized to a value (Y) specified by the sender in the delta field of a transmission-request block. For each packet except the last in a message, the send terminus sets a 32-bit *delta* field in the packet header to 1 and decrements its packet counter using 2's complement arithmetic. For the last packet it sets the delta field to the final value of the packet counter. It is easily seen that the sum of all delta fields in a transmitted message's packet headers equals Y, modulo 2^{32}.

In receiving interfaces, the delta field in a packet is added to its associated packet accumulator as the packet arrives. When the accumulator reaches zero again, the entire message has arrived and a receiving application can be notified (see section 2.6). If a packet delta field is zero, notification occurs without consulting the packet accumulator. This is used as an optimization for single-packet messages.

This deceptively simple mechanism provides two important capabilities:

- **Single message receive.** Out-of-order packet arrival is handled as described above. To summarize: if a message has only one packet, then its sender sets the packet header's delta field to 0 and notification occurs immediately upon arrival. If a message has several packets, then the sum of the packet headers' delta fields is 2^{32}. Since the receiving packet accumulator is also counting modulo 2^{32}, its value will return to 0 exactly when all packets have arrived.
- **Group receive.** A single notification can be generated when a set of messages from a known group of senders has arrived (e.g., for a distributed barrier operation). In this case, the *i*th sender is given an initial value Y_i such that the sum over all participating senders of $Y_i = 2^{32}$. This sum reaches 2^{32} and wraps around to zero exactly when all packets in the message set have arrived. Scatter-

gather I/O is a special case of this in which all messages come from the same sender.

Moreover, a process receiving a packet counter value Y_i can delegate work to (say) two other processes, *j* and *k*, as long as their initial packet counter values, Y_j and Y_k, sum to Y_i. The identity of a group's senders need never be known by a receiver. The Hamlyn library (section 3) provides a means for dividing buffer space and counter values among the delegates of the group.

Individual, multiple-packet messages could be handled using a simpler mechanism in which each packet header's delta field carries the total packet count, and a packet accumulator is reset by the first packet to arrive; however this would require the sending interface to calculate the total number of packets before sending any. Our scheme allows group reception and scatter-gather I/O while a receiver remains oblivious to the number of packets sent, and requires no extra memory.

2.6 Message arrival notification

A Hamlyn interface indicates that all packets of a message have arrived by appending an entry to a circular *notification queue* in main memory. There is one such queue for each receiving application. The interface then generates an interrupt if requested, the receiving process is asleep, and the processor has no pending interrupt requests from the interface for other processes. All this reduces interrupt overheads to minimum.

Notification queues are identified by *notification queue control blocks* (NQCBs) in the interface card's SRAM, which are in turn referred to by slot data structures (Figures 3 and 4). Each NQCB has a cursor that points to the tail of its notification queue; when all of a message's packets have arrived, an entry (Table 3) is written at the location indicated by the cursor and the cursor is advanced. When it advances beyond the queue's storage area, the cursor is reset and a wrap flag in the NQCB is toggled—that is, the notification queue is treated as a circular list.

Table 3: notification queue entry.

Wrap flag	Slot index	Notification index
Metadata entry pointer		
(padding for alignment)		

A receiver application wishing to busy-wait for a message maintains its own cursor and wrap flag and polls the next available notification queue entry until the wrap flags match. (The cache-coherent I/O of our prototype's workstation ensures that this busy-waiting causes no I/O bus activity until the entry is rewritten.) Because the notification queue is read-only by the application and

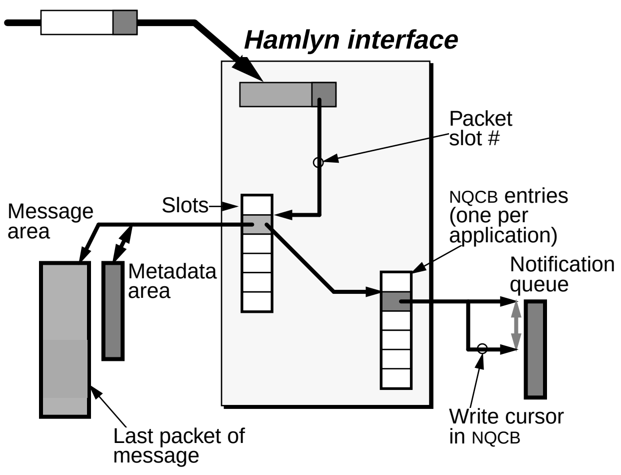

Figure 3: notification queues and message notification.

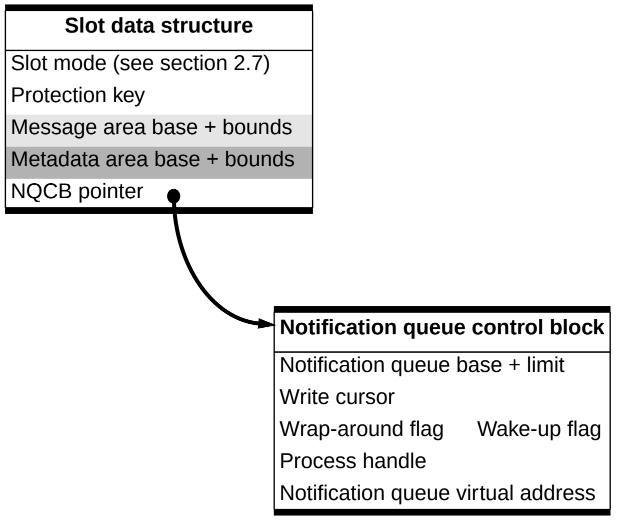
Slot data structure
Slot mode (see section 2.7)
Protection key
Message area base + bounds
Metadata area base + bounds
NQCB pointer
Notification queue control block
Notification queue base + limit
Write cursor
Wrap-around flag
Wake-up flag
Process handle
Notification queue virtual address

Figure 4: slot and notification queue control block contents. Several slots may point to the same NQCB.

write-only by the interface, no other locking or synchronization is necessary.

This is the only part of the Hamlyn design in which buffer overrun might occur: a faulty or malicious sender could transmit messages faster than a receiver can consume them, overwriting an older notification queue entry. This can be prevented by ensuring that the queue's size exceeds the number of messages arriving in the worst case. Section 2.7 describes a mechanism that can enforce an upper bound on this count; section 2.8 describes a discipline-based solution to bounding it.

2.7 Special modes of operation

The mechanism described so far accommodates cooperating processes in a single application, but we wanted Hamlyn to provide increased robustness, privacy, and security for client-server systems of mutually suspicious processes: specifically, several senders sharing a common receiver slot. To this end, slots can be put into two special modes of operation called paranoid and paranoid_one_shot. Both modes use extra fields in a message's metadata entry: a secondary base and bounds register, and a protection key that supersedes the slot data structure's key. When a packet arrives in a slot operating in one of these special modes, the packet header's base and bound fields are compared first to the slot's message area (primary) base and bounds, then to the metadata (secondary) base and bounds, while the packet header's key field is compared to the secondary protection key. All tests must pass before the packet is accepted. Additionally in paranoid_one_shot mode, no further use of the metadata entry is allowed without application software intervention after the first message for the entry has been received, preventing subsequent messages from being received before the first one is processed.

These modes provide several advantages:

- Senders' data can be confined to a small part of a message area, limiting the damage that a faulty or malicious client can cause.
- The paranoid_one_shot mode prevents a faulty or malicious client from overrunning a notification queue as long as the queue's size exceeds the number of metadata entries used. This mode also ensures that a packet's data cannot be overwritten after message arrival, so the data need not be copied elsewhere for safe-keeping.
- Every sender can have its own protection key, allowing revocation of one sender's access rights without affecting others. For example, if a node appears to have failed, all of its senders' keys can be revoked, potentially allowing message and metadata areas to be immediately reassigned.
- Since metadata areas are allocated in main memory, applications can change the secondary base, bounds, and key registers for their own metadata areas without OS intervention.

These modes entail more complicated interface logic and extra tests during packet arrival which introduce a small amount of extra latency. (There need not be more host memory accesses, since a metadata entry's packet accumulator must be updated anyway.)

A third special mode of operation, called fast mode, represents a special-case optimization for single-packet messages: if an arriving packet header's metadata index is all 1's, then the packet header is used to carry exactly one word of metadata, which is written in a notification queue entry in place of a metadata entry pointer. This lets a single-packet message carry a small amount of metadata with minimal overhead. In retrospect, it may have been a premature optimization.

2.8 Flow control and deadlock prevention

An important consideration in Hamlyn's design was to ensure that, in the absence of a failing sender, receiver, or interconnection fabric, data is never lost and communication never deadlocks. Deadlock is a treacherous issue. If Hamlyn employed low-level acknowledgment of each packet's transmission, and communications were ever blocked because the fabric cannot accept more packets, then acknowledgments could be blocked as well, potentially causing deadlock.

We avoid deadlock by not depending upon hardware packet acknowledgments and by arranging that the only hardware moderating the flow of incoming packets is the host's I/O bus. Incoming transfers are given priority over outgoing transfers for access to the I/O bus, so that the maximum time an incoming packet stalls at the interface is the time for one packet to traverse the I/O bus plus a small amount of overhead in the interface.

When power is first applied to a Hamlyn interface card, it enters a state in which it simply accepts and discards arriving packets. It does the same if the host fails to reset a periodic handshake timer. The Myricom switch used in our prototype detects powered-down interfaces and discards packets destined for them. It uses a round-robin service discipline for incoming ports to avoid starvation of incoming packets. If arriving packets are delayed while waiting for the Hamlyn interface to process them, or if there is conflicting traffic at a switch port, the switch and interface hardware generate link-level back-pressure, halting the sender. Eventually the sending application will block when its terminus' work queue fills up.

To streamline higher-level protocols, we exploit the fact that the interconnection fabric never loses packets in the absence of failures, which is a reasonable design decision for a small-area, multicomputer interconnection, although less so for a wide-area network. Sender management of memory automatically imposes a higher level of flow control for buffer space and metadata management, so a sending application blocks when no resources are available to service a request at the receiver.

A final concern is to prevent notification queue overrun. This can be accomplished by ensuring that the gap between the number of messages processed at the receiver and the number that can have been transmitted by the senders is always smaller than the number of metadata entries available. This limit can be enforced in paranoid_one_shot mode.

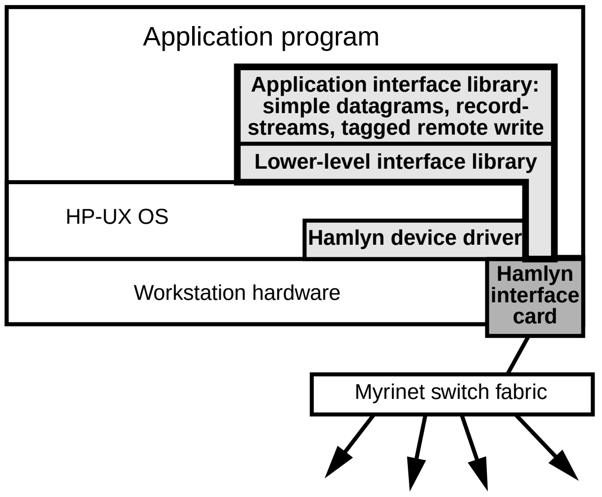
Application program
Application interface library: simple datagrams, record-streams, tagged remote write
Lower-level interface library
HP-UX OS
Hamlyn device driver
Workstation hardware
Hamlyn interface card
Myrinet switch fabric

Figure 5: schematic view of Hamlyn hardware and software. The shaded portions were added to support Hamlyn.

2.9 Architectural costs

Hamlyn consciously makes some design choices that impose additional costs compared to more traditional approaches:

1. Message, metadata, and notification queue storage areas must be wired down.
2. Applications are responsible for buffer management, including reclamation after sends and receives.

In return for these costs—which we think are relatively small, and not unlike those imposed by other high-performance network designs—Hamlyn provides fully-protected, direct application access to a network, automatic message segmentation and assembly, group reception notification, rejection of messages from failing or malicious processors, and both direct I/O and DMA sends.

3 The Hamlyn interface library

In order to make the Hamlyn architecture easier to use, we built a two-level application interface library. The upper layer provides a set of convenient programming abstractions and hides the details of buffer memory management. The lower layer provides a simple, efficient procedural interface to our communication hardware. The library was designed to provide a convenient infrastructure for popular middleware, such as MPI [Corbett95], Active Messages [vonEicken92], and Oracle's distributed lock manager. We describe these layers from the bottom up (Figure 5).

3.1 OS interface

The Hamlyn interface card is managed by a device driver module in the host operating system. OS modifications to support Hamlyn in UNIX systems[3] are largely confined to this driver, which provides all interface management services requiring OS mediation, such as creating slots and termini, installing slot protection keys, wiring and unwiring memory, and arranging to suspend or resume an application pending a message's arrival.

All other interface management services reside in unprivileged library code, linked in with application programs.

3.2 Low-level library procedures

The Hamlyn library includes a procedural interface to the network interface hardware and hardware-manipulated data structures. It uses a data structure called a *ticket*, which contains a message's destination, slot number, metadata index, protection key, data buffer base and bound, and some flags. Tickets are location-independent in that they may be exported in a message and used later to reply from a remote Hamlyn interface.

The library's lower layer has two main functions: h_send_msg and h_recv. The former accepts a ticket, a data buffer's address and length, and a metadata buffer's address and length. Buffer addresses are converted to offsets in sender message areas. A small message is written to the interface using direct I/O, while a DMA request is built for a large message. The function returns a handle that can later be used to decide when to release a buffer.

The h_recv function checks whether a message has arrived and, if so, it returns the address of the corresponding notification queue entry. A variant, h_recv_block, accepts a time-out argument and waits until either a message arrives or the specified interval expires.

3.3 Higher-level protocols

The Hamlyn library's upper layer supports a set of protocols with varying semantics to send and receive data. Three protocols, embodying most of its key ideas, are described below.

This layer was written in C++ because the language lets communication end points be represented as objects, keeping an application's name space clean and letting each protocol use the same operation names (*e.g.*, send, receive). We were inspired by the work of Stepanov and Lee on the C++ Standard Template Library [Stepanov95] in which aggressive inlining and optimization are combined to achieve highly efficient object code.

The library creates a Hamlyn manager (an instance of the hamlyn_manager class) for each send terminus. This class is responsible for managing the device driver interface and notification queues, and for allocating buffer memory.

[3] UNIX is a registered trademark of X/Open Company, Limited.

Applications create an end point for message reception by instantiating a receiver class, which contains one or more metadata areas and buffers, and a queue of received messages. It also provides a C++ virtual procedure (process_arrival) that is called by the Hamlyn manager to record a new message's arrival.

When receive is applied to an end point, a message is returned from the arrival queue in the receiver class instance if possible. Otherwise—if the queue is empty—the receiver instance calls a poll_nowait routine in the hamlyn_manager. The latter gets the next notification queue entry if there is one, follows its pointers to the metadata area and receiver end point, and then calls process_arrival there, passing it the address of the metadata. Control then returns to the receiver end point originally called by the application, which looks again for a new message. (It may not have received one if the notification queue was empty or the queue entry just processed represented a message for a different receiver instance.) This process continues until the original request is satisfied, a time interval expires, or the receiver end point chooses to block instead of busy-wait.

Although the Hamlyn architecture and the library's lower level are thread-safe, the upper level is not. Making it so remains a research topic.

All of the connection-oriented receiver classes support a make_seed call, which returns a *seed* object, containing tickets for preallocated buffers, metadata, and other information. A seed can be sent to a remote node in order to create an instance of the corresponding end point sender class. (It's so named by analogy with the similar purpose seeds serve in plants.)

The Hamlyn library uses these techniques to support the following protocols:

Simple datagram protocol. This provides access to raw Hamlyn hardware semantics. The send call is a wrapper for the h_send_msg routine. It creates a small amount of metadata that tells the receiver the offset and length of the transmitted data and can include a reply ticket. There is a separate receiver instance for each metadata entry, and each instance is either "ready" or "not ready": there is no queue.

Stream protocol. This provides a one-way, one-to-one, in-order connection from a sender to a receiver. The sender class supports send and flush. Large buffers are sent as-is, and small ones may be coalesced by copying. Calling flush forces transmission of all previously-posted data. The receiver class supports receive and release; the former returns a <start-pointer, length> pair describing its result; no copying occurs. It automatically allocates more buffers if needed. The release procedure frees all records up to and including that identified by its argument.

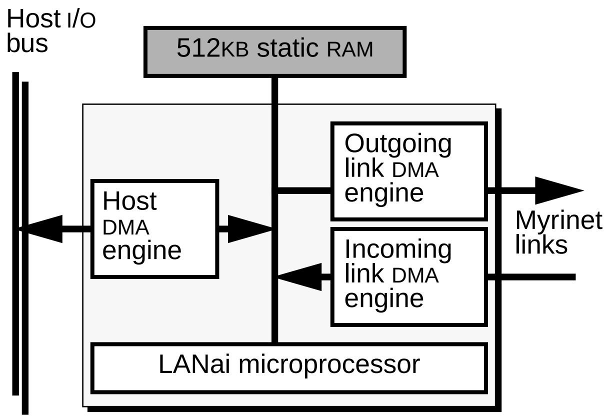

Figure 6: schematic view of the Myricom network interface.

Senders block if they ever get so far ahead of the receiver that they run out of tickets (which provide permission to write to a metadata entry).

Tagged Remote Write. In this protocol, a send call specifies a ticket, a source buffer's address and length, the destination buffer offset, and an integer tag. Tags are enqueued in the receiver and can be retrieved by calling get_tag or get_specific_tag. This protocol uses fast mode, so messages must fit in a single packet. In-order delivery is not guaranteed.

4 Performance evaluation

In order to evaluate the Hamlyn architecture, we collaborated with the University of California at Berkeley and Myricom, Inc., to build a prototype interface card for a Myrinet network [Boden95, Buzzard95].

The Myrinet switch we used is a non-blocking, 8x8 crossbar, which uses wormhole routing. It provides 80MB/S of bandwidth per port in each direction with about 0.5µs of latency. We used Myricom's LANai Version 4.0 network controller chip with 512KB of on-card static RAM, microcoded to implement the Hamlyn design. The LANai has a 32-bit CPU and three DMA units (Figure 6): incoming from the switch, outgoing to the switch, and to/from host memory. The host DMA engine is the only mechanism available to the LANai controller to access the host's main memory.

Our host computers were early-production HP 9000 Series 770 (J200) PA-RISC workstations with 100MHZ CPUs, running Version 10.00 of the HP-UX operating system. The interface cards plugged into the workstations' graphics I/O bus, which operates at the same frequency as the LANai CPU (40 MHz). The bus and its processor interface can support incoming DMA at 106MB/s, but outgoing transfers

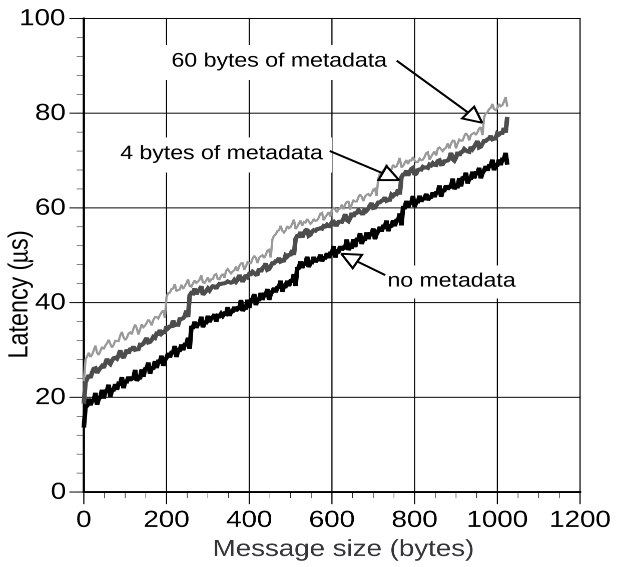

Figure 7: latency vs. message size for 256-byte packets.

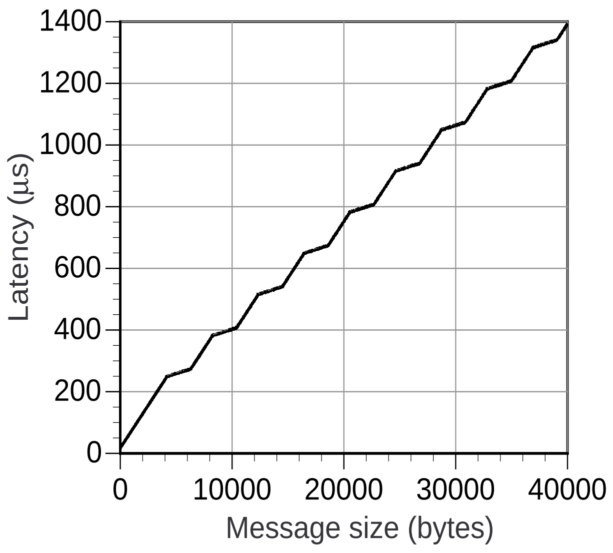

Figure 8: latency vs. message size for 4KB packets.

are limited to 32MB/s because of reduced opportunities for pipelining requests through the bus interface chip.

The workstations have a cache-coherent I/O architecture, which obviates the need for software to flush or purge data cache lines in DMA buffers. It also lets applications busy-wait for Hamlyn DMA completion without consuming I/O bus bandwidth. DMA buffers' I/O bus addresses are mapped to physical memory addresses by the workstations' memory-to-I/O bus interface hardware, so that our card can access a multiple-page buffer in a contiguous I/O bus address range—a considerable simplification.

Unless otherwise stated, our performance measurements were taken by sending a message from one application-level process to a second remote one, which returned the message to the sender. The round-trip times we measured were divided by two to get one-way data. Where we report single measurement numbers, we generated them by timing at least 10 000 messages. There was less than 1% variation between independent runs of our test suite.

The performance data reported here apply only to our test systems and do not necessarily represent products currently in production.

4.1 Short single-packet messages

The lowest latency is obtained using our interface's fast mode of operation (see section 2.7). We measured two cases using 16-byte payloads. In one, application code wrote data to the interface using direct I/O without the Hamlyn library, while the other case used the library's Tagged Remote Write protocol. The first case took 12.7µs one way (25.4µs round-trip), and the second case added 1.4µs of overhead, yielding 14.1µs in all (28.2µs round-trip). Table 4 shows where the time goes.

Table 4: one-way short-message transfer time (µs).

	raw interface	tagged-remote-write protocol
DMA	3.3	3.3
LANai	6.7	6.7
Switch	0.5	0.5
Host I/O writes	1.4	1.4
Host protocol software	0.8	2.2
Total	12.7	14.1

Notice that the 1.4µs due to the host I/O writes represents only 8 STORE instructions: these are slow because of the cost of traversing the I/O bus. By contrast the host protocol software costs represent tens or hundreds of instructions.

4.2 Latency for normal messages

Figure 7 shows the one-way latency as a function of message size with 0, 4, and 60 bytes of metadata and 256-byte packets. Several effects are visible:

- A baseline cost of about 17.4µs due to host software overhead, LANai control program overhead, and the cost of writing a notification queue entry.
- Increases in latency at message sizes that are multiples of 256 bytes. Our LANai code takes about 12.5µs more to receive a packet of this size than to send one, so the overall time increases at each packet boundary. The first step is larger than the others because the code changes from never updating the packet accumulator to updating it twice, while subsequent packets cause one update each.

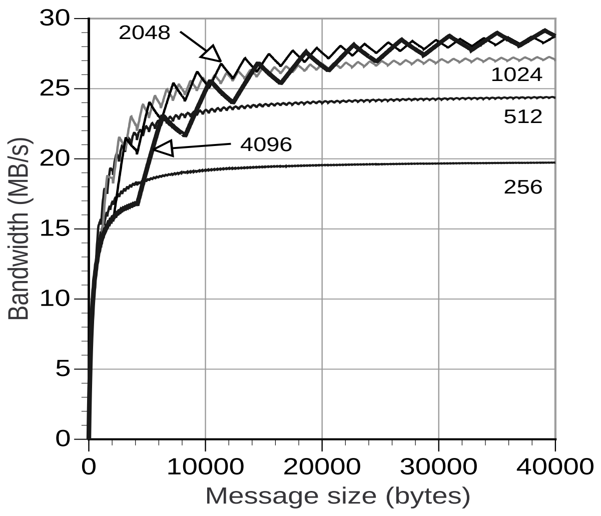

Figure 9: bandwidth vs. message size for packet sizes of 256, 512, 1024, 2048, and 4096 bytes and no metadata.

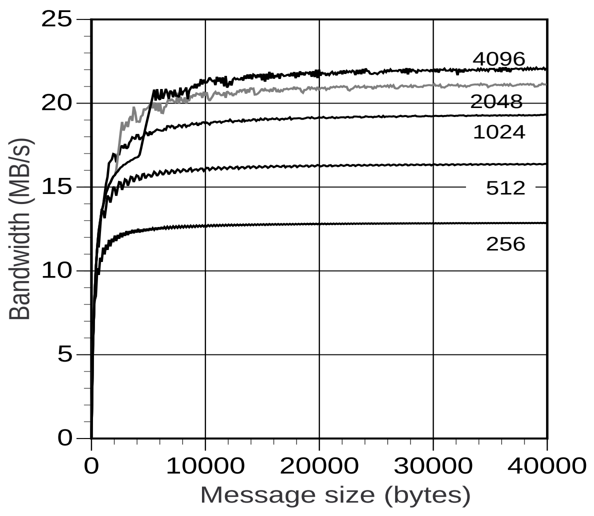

Figure 10: loopback bandwidth vs. packet size for packet sizes of 256, 512, 1024, 2048, and 4096 bytes.

- A marginal transmission cost of 55.5ns/byte (18.0MB/s). This results because the three steps involved are handled serially for each packet: about 31.3ns/byte to move outgoing data to the interface using DMA, 12.5ns/byte to move it across the network at 80MB/s, and 11.7ns/byte for DMA into the destination host.
- Repeating fine structure with a 32-byte period due to power-of-2 I/O bus transaction sizes. For example, a 28-byte transfer requires three transactions (for 16, 8, and 4 bytes) while a 32-byte transfer is done in just one.
- An extra 7.7µs to sending metadata, including low-level library overhead to translate buffer addresses, then start incoming and outgoing DMA. Metadata bytes incur the same transfer cost as other data but are not counted in the x-axis of Figure 7, so sending more metadata sent shifts the lines to the left.

Figure 8 shows the one-way latency for 4KB packets. Here, the bottleneck is moving data from the sending host to the Hamlyn interface card across the I/O bus controller. The latency exhibits alternating costs of 55.5ns/byte and 11.6ns/byte: if the last, partial packet of a message is almost full, each additional byte waits for outgoing DMA, transmission, and incoming DMA, totalling 55.5ns/byte. But small, final packets arrive early enough that they need only wait for the previous packet's DMA into the host memory to finish, which sets the marginal cost of 11.6ns/byte.

If a receiving process is asleep, latency is dominated by interrupt service and process-context switching time. We observed[4] 78µs for packets with no payload, which compares favorably with other recent reports [Jones96, Keeton95, vonEiken95]. (On the same machine, a context switch provoked by a semaphore takes 31µs.)

4.3 Bandwidth and packet size

Using 4KB packets, the bottleneck is the I/O bus interface. The observed slope of the latency function in Figure 7 is 30.7MB/s, from which we infer that the LANai control program achieves a 96% payload utilization of the 32MB/s outgoing DMA channel. The actual utilization is somewhat higher, since counter values are fetched and stored for each packet. (A potential optimization that we did not explore would be to cache some counters in the interface card.)

Our packet size can be altered by recompiling the LANai control program, so it is instructive to examine the latency achieved versus message size for various packet sizes. Figure 9 shows the overall one-way bandwidth as a function of message length for several packet sizes. The receiving interface takes 12.9µs per packet. For 256-byte packets, this limits bandwidth to 20MB/s, which agrees with our observations. Asymptotic performance increases with packet size, but 2KB packets outperform 4KB packets for smaller messages because there is more concurrency between outgoing DMA, transmission, and incoming DMA. The advantage of 4KB packets is slight and inconsistent up to message sizes of 40KB, the largest we measured.

Even though the interface does not pad packets to their maximum size, there are conflicting pressures on the choice of packet size:

- Choosing a packet size large enough that network transit time exceeds LANai packet-processing time

[4]. Our prototype had a defect, which was impractical to fix, that caused the actual times to vary widely. This was the minimum latency, which we are confident would be the latency in a corrected system.

keeps the LANai from being a bottleneck. The larger the packet size, the more LANai time remains for other tasks, including processing packets moving in the opposite direction.

- Long delays should be avoided. Transferring a 4KB packet occupies a link and a DMA channel for about 50 µs, a long time to block another packet needing the same resources. This argues in favor of shorter packets.

Taking these into account, we recommend a packet size of 1KB for the particular combination of bandwidths and overheads measured for our prototype.

The effect of competing for LANai CPU cycles and I/O bus bandwidth can be seen when a message's source and destination are the same process on the same host, as shown in Figure 10. With only 4 bytes of data, the latency in this loopback test rises to 19.1 µs because sending and receiving compete for the same LANai controller chip. The bandwidth figures tell a similar story: the asymptotic bandwidth of 28MB/s from Figure 9 drops to 22MB/s in Figure 10 because of I/O bus contention.

4.4 Projections for alternative hardware

The LANai performance is relatively low because it is a programmable controller. If we were to implement a Hamlyn interface using hardware state machines, we estimate that the one-way, application-to-application short-message transfer time would decrease to about 6µs, but the large-message bandwidth, which is limited by the I/O bus, would not change appreciably.

5 Related work

There has been a great deal of work in the field of interface design for high-speed interconnections, especially since the original Hamlyn design was written up. The history of these ideas is not entirely clear: several teams were inventing similar-sounding approaches at around the same time. It is neither fair to say that Hamlyn copied from them, nor that they copied from Hamlyn. (For the record, the earliest extant reference to Hamlyn is dated July 1992.) Although we think that Hamlyn's main contribution lies in its coupling of sender-based memory management to its protection scheme, we present a somewhat broader summary of what we consider to be the work most relevant to our finished design.

IBM's OS/360 provided variants of put and get file-system calls that avoided data copying by having the OS specify the location of the buffer to use, rather than the application [Clark66, Belady81]. Hamlyn uses a variant of this mechanism in its interface library.

5.1 Load/store interfaces

A STORE instruction can be thought of as a degenerate, sender-directed message—indeed, there is a large and active literature that views large-scale shared memory machines in this way, of which the Cray T3D, Convex Exemplar, KSR AllCache architecture, Alewife [Kranz93], Typhoon [Reinhardt94], and low-level SCI protocols [IEEE92] are representative examples. All require dedicated hardware support that is tightly integrated with the host processors.

Several groups have used the LOAD/STORE paradigm in less tightly coupled systems to provide an interface to cross-network communication. For example, the Alto remote memory reference protocol [Spector81] used network messages in this way; [Thekkath94] discusses the idea of separating data movement from notification in remote LOAD/STORE operations (Hamlyn also allows this); and SHRIMP [Blumrich94] provides low-latency remote-memory access using hardware support for automatic data replication, coupled to a virtual-memory protection scheme. Many of these schemes provide excellent performance for the particular operations that they support—specialization is a powerful tool for lowering latency—but sometimes at the expense of relatively high processor utilization. Most implicitly depend upon in-order packet delivery.

The hybrid deposit model [Osborne94] combines sender-based addressing with the execution of small programs on a remote node, using both local and remote data—a considerable generalization of the *remote fetch-and-op* proposed in [Wilkes92]. Implemented in software on top of a 155Mb/s ATM system, it achieved a round-trip time of 49µs without a switch and 60µs with one. Osborne credits [Subhlok93] with introducing the term "deposit model" for what we call sender-based memory management.

5.2 Copy avoidance

Several projects have used page-remapping and smart interface buffer allocation to accelerate processor-to-interface communication, including the *fbufs* work at the University of Arizona [Druschel93], the Medusa FDDI interface [Lumley92, Banks93] and the follow-on Afterburner project [Dalton93].

The Nectar system [Cooper90] allowed applications direct access to its communication interface memory in order to eliminate copies at the cost of all accesses being to memory in the I/O space. It achieved round-trip RPC latencies of 500µs across a 100Mb/s network.

ATM network interfaces can use virtual circuit identifiers (VCIs) to provide early demultiplexing of incoming data to user data buffers. One such use occurred in the Osiris project [Druschel94], which combined stream demultiplexing using ATM VCIs into fbufs, some support for out-of-order delivery, and direct access to the network interface for a limited number of applications. Together, these achieved a round-trip latency of 154µs and a maximum throughput of about 41MB/s on a 622Mb/s ATM network.

CMU's Hardware-Assisted Remote Put (HARP) interface to the CreditNet ATM adapter card allows applications to send directly from their own buffers (akin to Hamlyn message areas), and to provide a set of buffers into which data for a virtual circuit is placed, but it does not appear to allow direct addressing of remote memory on a per-message basis [Mummert96]. We were unable to locate any published latency figures for this interface.

5.3 Cranium

Cranium [McKenzie94], like Hamlyn, was designed to provide a host interface to a packet-switching fabric that performed adaptive routing. Like Hamlyn, it has message areas that are used to send and receive data, although it appears that these are restricted to 2KB pages, and the expectation is that there is a single message per area, since there seems to be no provision for a message-offset field. Multiple packets use a sequence number to allow reassembly, rather than offsets; this simplifies the hardware, but it requires that all packets in a message be of the same size. (This means that Cranium could not handle variable-length metadata.) Receivers specify the identity of senders expected to write to a message area, and this is used for protection checks. (There is no discussion of how spoofing is prevented.) Cranium also provides queueing channels, which allow messages to be appended to the end of a message area. We thought about providing these for Hamlyn but eventually decided not to: (1) to force us to work through all of the details of pure, sender-based memory management; (2) to avoid introducing a *prima facie* source of receiver buffer overruns; and (3) because such messages have to be restricted to single packets. Cranium supports many of the goals of Hamlyn, but its designers made several decisions to reduce functionality in order to simplify the interface. It thus represents a different point in the design space.

5.4 Active Messages

Active Messages [vonEicken92, vonEicken94, Martin94] provide a set of arrival semantics for single-packet messages by including the address of a function to call in each one. The function is typically invoked in a restrictive environment on the interrupt stack, with no protection barriers around it. As a result, aggressive implementations of Active Messages are the standard performance target for this kind of work. The main difference is that Hamlyn provides security between applications; data placement is controlled by the sender, rather than the receiver; and all protocol processing happens in a well-defined application context. "Although the restrictions and limitations of previous interfaces [to Active Message systems] made their implementations simple and efficient, the same restrictions and limitations prevent them from supporting the broader spectrum of applications now required" [Mainwaring95a].

[Karamcheti94] reported instruction counts (but no timings) for Active Messages on a *CM-5* (CMAM), which are roughly comparable to ours although they were measured on a SPARC processor and ours are for PA-RISC. The CMAM finite-sequence, multiple-packet delivery protocol seems to provide functionality that approaches our simple datagram protocol: it does not support our group-receive operations, but, like ours, it does handle out-of-order packet delivery. [Karamcheti94] quotes 397 instructions to do a 16-word (64-byte) unidirectional send. A send using Hamlyn's tagged remote write consumes 260 processor cycles (fewer instructions), most of which are consumed when the processor stalls while writing to the I/O bus, and a receive consumes 120 cycles.

[Wallach95] reports the lifting of one of the restrictions on Active Messages: that the handlers must not block.

We think that the idea of Active Messages is good, and we are gratified that some Hamlyn features are making their way into a revised proposal [Mainwaring95a], which supports protection, caching end point descriptions,[5] as well as multiple send and receive areas per end point.

On the other hand, we think that a scheme requiring host processor intervention on every packet would **not** be such a good idea because the process-context switches would prove too expensive. Indeed, the current trend in processor design seems to be toward ever-larger amounts of machine state, which will make this more costly still. Hamlyn addresses this concern by automating message reassembly in the interface card.

U-Net [vonEicken95] embodies some of the same principles as Hamlyn, including direct user-level access to the interface in order to eliminate OS involvement whenever possible, and end points that can route incoming messages directly to application memory. Like Hamlyn, the prototype U-Net implementation is built by re-microcoding an existing interface card—a Fore Systems ATM interface. (By their definition, Hamlyn is using a "standard network interface"!) Since it is built on ATM, which is inherently unreliable, U-Net has to deal with lost packets. Its performance is slightly worse than Hamlyn's: [vonEicken95] reports Active Message round-trip times on top of U-Net of 79µs for 32 bytes of data or

5. [Wilkes92] suggested the same idea as a way to conserve interface-card memory.

less (41µs of which is due to the ATM switch; the equivalent Myrinet time is 1µs) and 135µs + 0.2µs/byte for bulk data transfers. (The equivalent Hamlyn numbers are probably the 28µs for a round-trip tagged remote write, and the one-way bulk data transfer cost of 17.4µs + 32.6ns/byte with 4KB packets.)

6 Conclusions

What did we learn from this exercise? First, the basic Hamlyn approach seems to have been validated: we can provide low-latency, high bandwidth, protected communication directly from multiple application programs, with little or no OS intervention. We also picked up a few other observations and lessons along the way.

6.1 Network demands

[Karamcheti94] argues that the underlying network should provide in-order delivery, deadlock freedom, and fault-tolerant packet transmission. We conclude instead that Hamlyn can synthesize in-order delivery cheaply, assuming a deadlock- and error-free network, giving interconnection designers freedom to optimize for performance, rather than high-level protocol support. [Davis92] argues that an adaptive-routing network can achieve roughly twice the throughput of a non-adaptive one.

6.2 Buffer management

Hamlyn does not copy outgoing messages, so applications must be coded to avoid reusing buffers until transmission is complete. To help with this, the Hamlyn library provides a function that determines whether a message buffer can be reclaimed.

Metadata often originate in a few small variables on an application program's run-time stack, but if transmission is done using DMA, they must be copied to a special, wired-down metadata area. This proved burdensome; if we were to redesign Hamlyn, we might always send metadata using direct I/O.

6.3 Opening and monitoring connections

There is a "bootstrapping" problem when contacting a long-lived server: a potential client cannot transmit to the server because it has no resources allocated there, and the server cannot send a ticket or seed because it does not know of the client's existence. We considered adding an unreliable FIFO message queue, but we decided that since these operations are not time-critical, they could be done with standard OS services, which might themselves use Hamlyn inside the operating system. (The lowest-level bootstrapping problem here can be solved by allocating well-known slots, one for each remote processor, which the OS instances can use to establish higher-level communication paths.)

A similar observation applies to detecting peer-process failures. We once thought that an "Are you there?" message should be sent periodically between processes in a highly available system, but if such a polling interval expires without a reply, an application does not know whether the system is overloaded, or the polled host has failed, or a peer process is dead, or the process is stuck in a long computation. On the other hand, the OS has definitive knowledge of process' states and so can prevent much of this confusion. The moral is to let the OS do what it is good at.

6.4 Closing connections

Traditional networks have difficulty providing reliable connection close because of potential message loss. In the absence of such loss, they can do a good job because the OS knows about the connection setup and can tear it down even after the application dies. (This is even true in most application-level protocol suites, which invoke the OS for connection setup/teardown.) In Hamlyn, the OS cannot fulfill this role because it has no knowledge of the connections, so we reverted to a model where our prototype only allows graceful close operations by a sender.

6.5 Stronger security

Hamlyn uses 64-bit protection keys. We estimate that our prototype can detect and discard a packet having an invalid key within 3µs. At this rate, a brute-force attack is likely to take about 877,000 years. Keys can be generated using cryptographic-quality pseudo-random number generators, or generators embodying true random processes, so we think that attack by guessing keys is futile. But Hamlyn keys reside in applications' memory address spaces, so our defense against forged messages depends upon memory privacy. For this reason, and because some users find probabilistic protection unsatisfactory, we thought briefly about other techniques:

It would be easy to make protection keys accessible only indirectly, and have applications specify indices into a secure, per-process table of keys, maintained by the OS. A secure OS would then make keys unforgeable. But this scheme would fundamentally alter the Hamlyn paradigm, since almost all communication channel management would then require OS intervention.

In practice, this is largely irrelevant because the main security problem is user passwords, which are much simpler to attack than 64-bit binary keys.

6.6 Interface memory cost

The original Hamlyn design proposed that slots be cached by interface hardware in order to make them abundant and cheap. Our prototype allocates slot data structures in expensive interface card SRAM, and metadata areas in main memory. We never satisfactorily established the right trade-off between function and complexity here.

6.7 Summary

The Hamlyn architecture provides users with a message passing interface having a combined hardware and software latency of just a few microseconds, while providing full protection between mutually suspicious applications. We described the most important techniques underlying our implementation, including design trade-offs that we can make (and have made), and we presented performance measurements.

Our design is optimized for closely-coupled, multicomputer systems. It yields better performance than loosely-coupled clusters of autonomous computers and, due to the inherent isolation of message-passing systems, provides much better fault tolerance than shared-memory systems, as well as inter-application protection at low cost. All of these needs must be addressed if large-scale, parallel machines are to have a significant impact upon general-purpose computing. The Hamlyn architecture is an important step in that direction.

Acknowledgments

Martin Fouts and Bill Worley of HP Laboratories provided the initial encouragement to turn Hamlyn from a random thought into a proper architecture. Monroe Bridges and his colleagues in Hewlett–Packard's Networked Computing Division provided valuable insight from the perspective of product designers, forcing us to justify and simplify our design. Cedric Krumbein and other members of the Network of Workstations project in the Computer Science and Engineering Division of the University of California at Berkeley helped us to design our interface cards, while the staff of Myricom, Inc., helped us to build them and gave us a considerable amount of advice, sample firmware, and early access to their technology, enabling our success.

References

[Banks93] D. Banks and M. Prudence. A high performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications* **11**(2), February 1993.

[Belady81] L. A. Belady and R. P. Parmelee, and C. A. Scalzi. The IBM history of memory management technology. *IBM Journal of Research and Development*, **25**(5):491–503, September 1981.

[Blumrich94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felton, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL). Published as *Computer Architecture News*, **22**(2):142–53. ACM/IEEE, April 1994.

[Boden95] Nannette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles E. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: a Gigabit-per-second local area network. IEEE *Micro*, pages 29–36, February 1995.

[Buzzard95] Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes. Hamlyn: A high-performance network interface with sender-based memory management. Presented at *HotInterconnects III* (Stanford, CA), August 1995. Available from http://www.hpl.hp.com/personal/John_Wilkes /ftp-index.html#Hamlyn.

[Clark66] W. A. Clark. The functional structure of OS/360: part III, data management. *IBM Systems Journal*, **5**(1):30–51, 1966.

[Cooper90] Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol implementation on the Nectar communication processor. *Proceedings of the ACM SIGCOMM'90 Symposium* (Philadelphia, PA), September 1990.

[Corbett95] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. *3rd Annual Workshop on I/O in Parallel and Distributed Systems (IOPADS'95)* (Santa Barbara, CA), pages 1–15, April 1995.

[Dalton93] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network* **7**(4):36–43, July 1993.

[Davis92] Al Davis. Mayfly: a general-purpose, scalable, parallel processing architecture. *Lisp and Symbolic Computation* **5**(1–2):7–48, May 1992.

[Druschel93] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC), pp 189–202, December 1993.

[Druschel94] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: a software perspective. *Proceedings of the 1994 ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications*, pp 2–13, August 1994.

[IEEE92] Standard for scalable coherent interface (SCI). IEEE Standard 1596-1992.

[Jacobson95] David M. Jacobson. *Method and apparatus for determining when all packets of a message have* *arrived.* US patent application, filed 24 February 1995, allowed 25 June 1996.

[Jones96] Rick Jones. *NetPerf.* http://www.cup.hp.com/netperf/NetperfPage.html, Hewlett-Packard Company, Cupertino, CA.

[Karamcheti94] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: where does the time go? *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA). ACM, October 1994.

[Keeton95] Kimberley Keeton, Thomas Anderson, and David Patterson. LogP quantified: the case for low-overhead local area networks. Presented at *HotInterconnects III* (Stanford, CA), August 1995. Available at http://now.cs.berkeley.edu/Papers/Papers /hotinter95-tcp.ps.

[Kranz93] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: early experience. *Proceedings of 4th ACM Annual Symposium on Principles and Practice of Parallel Programming*, May 1993.

[Lumley92] J. Lumley. *A high-throughput network interface for to a RISC workstation.* Hewlett-Packard Laboratories technical report HPL–92–7, January 1992.

[McKenzie94] Neil R. McKenzie, Kevin Bolding, Carl Ebeling, and Lawrence Snyder. Cranium: an interface for message passing on adaptive routing networks. *Proceedings of Parallel Computer Routing and Communication Workshop* (Seattle, WA), pages 266–80, May 1994.

[Osborne94] Randy Osborne. A hybrid deposit model for low overhead communication in high speed LANs. *Proc. 4th Intl. IFIP Workshop on Protocols for High-speed Networks*, August 1994. Available as http://www.merl.com/TR/TR94-02c/Welcome.html.

[Reinhardt94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Typhoon and Tempest: user-level shared memory. *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL). Published as *Computer Architecture News*, **22**(2):325–36. ACM/IEEE, April 1994.

[Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, **2**(4):277–88, November 1984.

[Stepanov95] Alexander Stepanv and Meng Lee. The Standard Template Library. Technical Report HPL-95-11 (R.1), Hewlett-Packard Laboratories, 1995.

[Subhlok93] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA), May, 1993, pp 13-22.

[Thekkath94] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating data and control transfer in distributed operating systems. *Proceedings of ASPLOS VI* (4–7 Oct. 1994, San Jose, CA). Published as *Operating Systems Review* **28**(5):2–11, December 1994.

[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, May 1992.

[vonEicken94] Thorsten von Eicken, Veena Avula, Anindya Basu and Vineet Buch. Low-latency communication over ATM networks using active messages. Presented at *Hot Interconnects II* (Stanford, CA), August 1994.

[vonEicken95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net, a user-level interface for parallel and distributed computing. *Proceedings of the ACM Symposium on Operating System Principles* (Copper Mountain Resort, Colorado). Published as *Operating Systems Review* **29**(5):40–53, December 1995.

[Wilkes92] John Wilkes. *Hamlyn—an interface for sender-based communications*. Technical report HPL-OSR–92–13. Operating Systems Research Department, Hewlett-Packard Laboratories, Palo Alto, CA, November 1992. Available from http://www.hpl.hp.com/personal/John_Wilkes /ftp-index.html#Hamlyn.

[Wilkes95] John Wilkes. *Inter-processor communication system in which messages are stored at locations specified by the sender.* US patent number 5,448,698, granted Sept. 1995.

The authors can be contacted as follows: gdb@geoplex.com, {jacobson, marovich, mackey, wilkes}@hpl.hp.com. Please address correspondence to David Jacobson.