Analyzing Communication Latency using the Nectar Communication Processor

Peter Steenkiste

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

For multicomputer applications, the most important performance parameters of a network is the latency for short messages. In this paper we present an analysis of communication latency using measurement of the Nectar system. Nectar is a high-performance multicomputer built around a high-bandwidth crosspoint network. Nodes are connected to the Nectar network using network coprocessors that are primarily responsible the protocol processing, but that can also execute application code. This architecture allows us to analyze message latency both between workstations with an outboard protocol engine and between lightweight nodes with a minimal runtime system and a fast, simple network interface (the coprocessors). We study how much context switching, buffer management and protocol processing contribute to the communication latency and we discuss how the latency is influenced by the protocol implementation.

1. Introduction

Multicomputers that use existing hosts and a general network are an attractive architecture for many applications [11], and they are one of the main motivations for improving the performance of networks [1]. Some users are interested in partioning applications across a large number of workstations, while other users want to combine the resources of a smaller number of supercomputers. In order to make these general multicomputers a viable architecture for a wide range of applications, networks have to support low latency and high bandwidth communication.

This research was supported in part by the Defense Advanced Research Projects Agency (DOD) monitored by DARPA/CMO under Contract MDA972-90-C-0035.

Reducing latency has traditionally been the biggest challenge. Dedicated multicomputers with special-purpose interconnects such as the Intel Touchstone system have latencies below 100 microseconds [4], while latencies between Unix workstations communicating over general networks are typically one order of magnitude higher. The latency between two Sun4/330 running Sun OS 4.1 is for example about 800 microseconds. One reason for the higher latency is the difference in communication medium. The interconnection networks used by dedicated multicomputers only have to cover a few meters and guarantee data integrity in hardware, while general networks have to cover much larger distances and introduce errors in the data stream with non-zero probability. The communication protocols that recover from these errors introduce overhead, thus adding to the latency. This overhead however is, or should be, of the order of 10s of microseconds [7], and does not account for the order of magnitude difference in latency. In this paper we identify other sources of overhead based on measurements collected on Nectar.

The Nectar network [3, 11] consists of a high-bandwidth crosspoint network (Figure 1-1). Host are connected to the network through Communication Acceleration Boards (CABs) that are responsible for protocol processing. A 26-node prototype system using 100Mbit/second links has been operational since 1989 and has been used to parallelize several engineering and scientific applications [12]. The host-host latency over Nectar is about 200 microseconds. One of the goals of the prototype was to experiment with different protocol implementations and make the communication coprocessor customizable by applications. For this reason, we built the CAB around a general-purpose CPU with a flexible runtime system [8]. The application-level CAB-CAB latency is about 150 microseconds.

The Nectar architecture allows us to study the communication between different types of hosts:

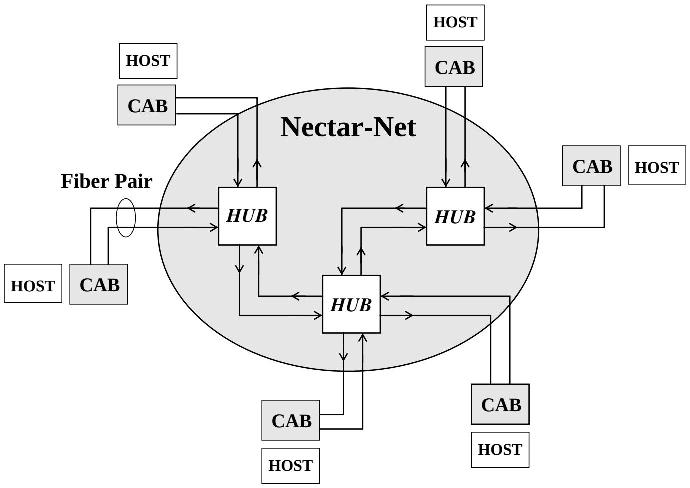

Figure 1-1: Nectar system overview

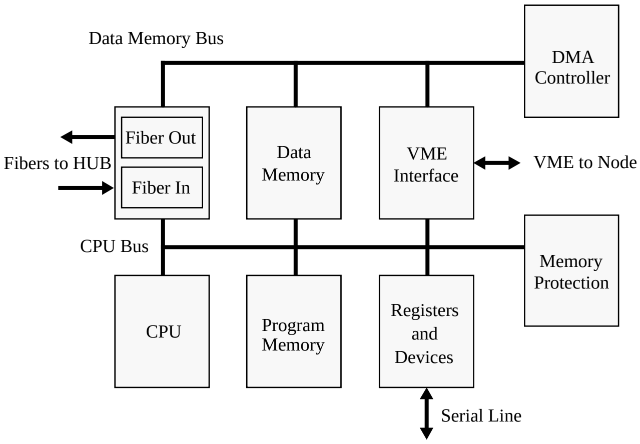

Figure 2-1: CAB architecture

traditional workstations with a powerful outboard protocol engine and ‘‘light-weight’’ hosts with basic network interface, i.e. the CABs. The CAB runtime system is similar to the runtime system on dedicated multicomputers and to micro-kernel operating systems.

In this paper we first give an overview of Nectar, concentrating on the CAB architecture and Nectar communication software (Section 2). In Section 3 we present a breakup of the CAB-CAB message latency for a number of communication protocols, and we discuss how these overheads are influenced by the CAB hardware and runtime system, and the protocol implementation. Section 4 analyzes the host-host message latency. We also present estimates for how the host-host latency would change if we replaced the flexible CAB by a hardwired protocol engine. In Section 6 we discuss related work.

2. The CAB architecture and software

The CAB is built around a 16.5 MHz SPARC CPU [17] and devices such as timers and DMA controllers (Figure 2-1). It is connected to the host through a VME bus. To provide the necessary memory bandwidth, the CAB memory is split into two regions: a working memory for the SPARC (program memory), and a packet memory (data memory). DMA transfers are supported for data memory only but the SPARC can access both memories equally fast. The memories are built from static RAM and there is no cache. The memories are directly accessible to applications on the host.

The SPARC runs a flexible runtime system [8] that provides support for multiprogramming (a threads package) and for buffering and synchronization (the mailbox module). Several communication protocols have been implemented on the CAB using these facilities: the Nectar-native datagram, reliable message (RMP) and request-response (RR) protocols, and the standard internet protocols (UDP/TCP/IP). We describe the software in more detail in the remainder of this section.

2.1. Threads and SPARC register windows

Previous protocol implementations have demonstrated that multiple threads are useful, but multiple address spaces are unnecessary [6, 13, 15]. As a result, we designed the CAB to provide a single physical address space with a runtime system that supports multiple threads. The threads package is based on Mach C threads [9]: it provides mutex locks to enforce critical regions, and condition variables for synchronization. Preemption and priorities were added so that system threads (e.g. protocol threads) can be scheduled quickly, even in the presence of long-running application threads [8].

The SPARC CPU has seven register windows and eight global registers. Each window has eight input registers, eight output registers, and eight local registers; the input registers of one window overlap with the output registers of the next window. Register windows support fast procedure calling and returning: during a call, the processor switches to the next window and it switches back when the call returns, so typically no registers have to be saved or restored. Registers only have to be saved and restored on window overflow and underflow, i.e. when the call depth exceeds the number or register windows.

The threads packages utilizes the SPARC register windows in a fairly standard way. The executing thread can use all the windows in the register file. On a thread context switch, the entire processor state, including the register windows that are in use, is flushed to main memory, and the state of the next thread, including the contents of the top window, is restored. When a thread blocks and no other threads are runnable, it polls the thread ready queue; if it is also the first thread to be woken up, no context switching is needed.

When a trap happens, the SPARC changes the current window pointer automatically to the next window, and the program counters are placed in local registers of that window. This allows traps to be handled without saving any registers, but since the ‘‘next’’ window is always reserved for traps, threads can only use six windows. Traps build their stack on top of the stack of the executing thread, so the trap handler shares the register window stack with the thread that was running at the time of the interrupt.

2.2. Mailboxes

Mailboxes are queues of messages with a network-wide address. Host processes and CAB threads communicate over Nectar by sending messages to remote mailboxes using a transport protocol, while a host and its local CAB can exchange messages through a mailbox directly. Mailboxes provide synchronization between readers and writers. For example, a host process can invoke a service on the CAB by placing a request in a server mailbox; this wakes up the server thread that is blocked on the mailbox. Mailboxes form a uniform mechanism for both intra-node and inter-node communication.

The primitives operations on mailboxes are placing a message in a mailbox (**Put**) and fetching a message from a mailbox (**Get**). Both operations can be executed in two steps to avoid the copying of data. **Begin_Put** and **Begin_Get** return a pointer to an empty buffer or a message in the mailbox. The user can then build or consume the message in place. **End_Put** and **End_Get** return the message or empty buffer to the system. The buffer space for the messages in a mailbox is allocated in CAB memory. By mapping CAB memory into their address spaces, host processes can build and consume messages in place.

2.3. Communication over Nectar

Figures 2-2 and 2-3 show the path that is taken through the software when a message is sent between two Nectar CABs or hosts. Vertical arrows indicate procedure calls and horizontal arrows indicate context switches, either between threads or between a thread and interrupt handler.

Application threads on the CAB send messages by executing a procedure call to the desired transport protocol

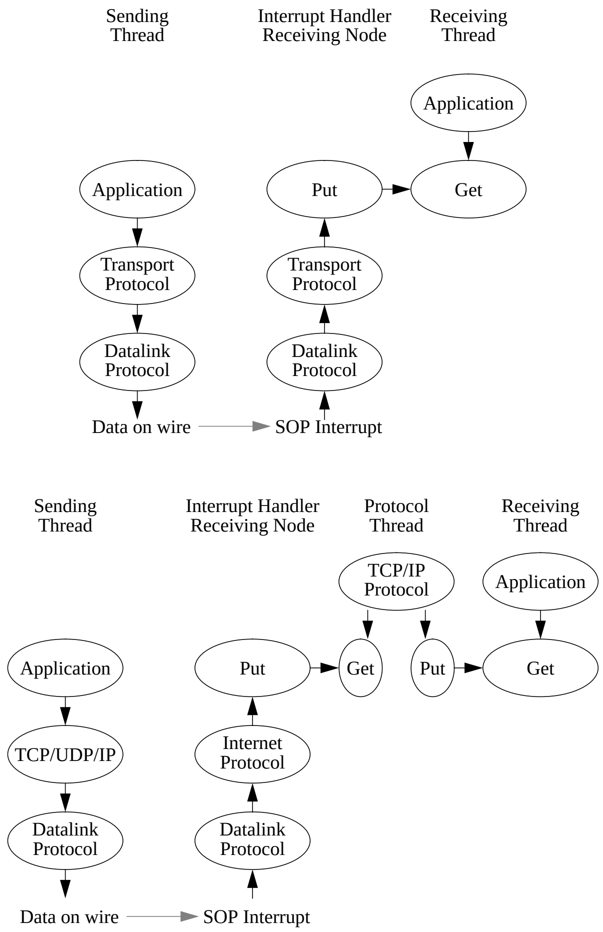

Figure 2-2: CAB-CAB communication with Nectar-native (top) and Internet (bottom) protocols

(Figure 2-2). The transport protocol does the necessary processing, hands the packet off to the datalink protocol, which places it on the wire. When the message arrives on the receiving CAB, the SPARC is interrupted, and the datalink protocol places it in a mailbox using **Begin_Put**. The transport protocol performs the matching **End_Put**, which wakes up any waiting application thread or process.

The transport protocol can be called in one of two ways. The Nectar-native protocols are invoked by an upcall inside the interrupt handler [6] (top Figure 2-2). This implementation was motivated by speed: it avoids waking up a thread. The disadvantage is that since many data structures are accessed at interrupt time, critical regions often have to be enforced by masking interrupts. Furthermore, by doing a lot of processing at interrupt time, the CAB response time is potentially high. An alternative organization is to move the transport protocol processing from the interrupt handler to a thread. This simplifies the software organization, and possibly improves response time. This organization was chosen for the internet protocols (bottom Figure 2-2).

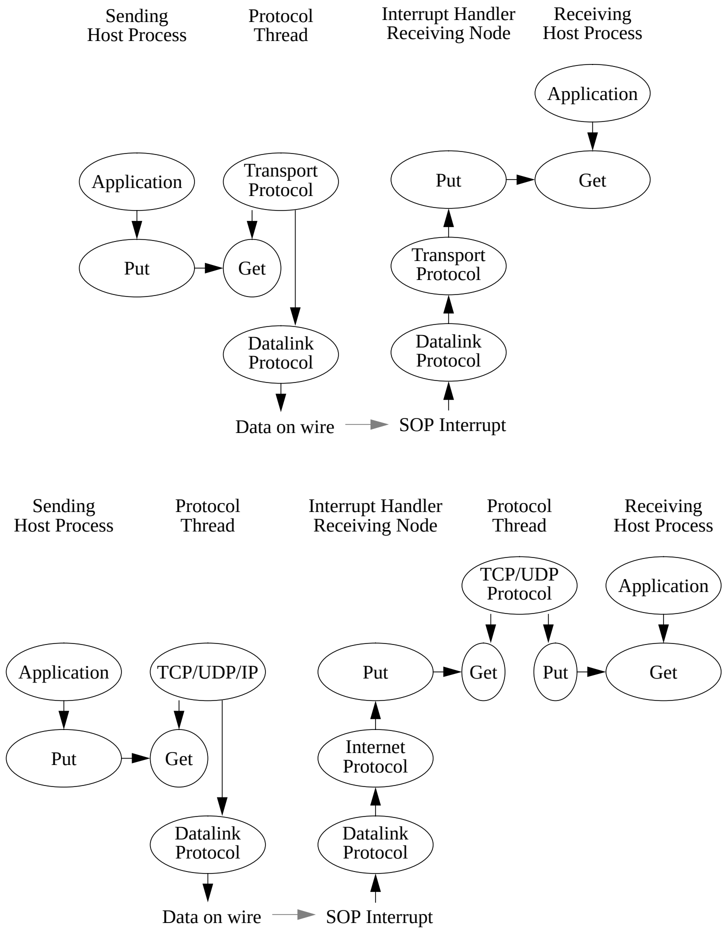

Figure 2-3: Host-host communication with Nectar-native (top) and Internet (bottom) protocols

Host processes send message over Nectar by placing a request in a protocol mailbox in CAB memory (Figure 2-3). This wakes up a protocol thread on the CAB, which performs the send, and then blocks on the mailbox, waiting for the next send request. The behavior of the sending protocol thread is similar to that of a sending application thread in CAB-CAB communication. Incoming messages are handled in the same way as was described for the CAB-CAB test, except that the message is read from the mailbox by a host process. Host processes that are waiting for a message have the option of polling for a short period of time, thus (potentially) avoiding the overhead of being put to sleep by the Unix scheduler.

Table 2-1 shows the CAB-CAB and host-host latency for the Nectar-native and UDP protocols. The results are half of the roundtrip time between two application processes or threads, and they include the time to build and consume a one-word message. We analyze these results in the remainder of the paper. We briefly look at performance measures other than latency in Section 5.

Protocol	CAB-CAB Latency	Host-host Latency
Datagram	97	169
Request-Response	154	225
RMP	127	213
UDP	234	308

Table 2-1: Message latency in Nectar (microseconds)

3. CAB-CAB breakup

The CAB architecture and runtime system support the execution of application code on the CAB. Although this was not the intent, the collection of CABs can be viewed as a light-weight distributed-memory multiprocessor. Studying CAB-CAB latency in this CAB multicomputer gives us insights into the network performance we can expect in multicomputers using general networks and nodes with a fast but simple network interface.

Table 3-1 breaks up the CAB-CAB latency measurements. The cost is shown for interrupt processing, buffer management (mailboxes), protocol processing (datalink and transport), condition handling (wait and broadcast), thread management and register window overhead. The measurements were collected using a functional simulator of the CAB. This simulator is built around a SPARC simulator provided by Sun, and its performance matches the performance of the hardware within 0.5%. Multiple CAB simulators can be linked together to simulate a multi-CAB network.

	Dgram	RR	RMP	UDP
Send				
Datalink protocol	14	12	12	14
Transport protocol	6	7	22	23
Mailbox put	0	8	0	8
Other	2	9	2	0
Receive				
Interrupt handling	7	7	7	7
Datalink protocol	18	24	44	23
Transport protocol	0	18	9	27
Mailbox put + get	28	35	25	52
Conditions	10	10	0	21
Thread switch	0	0	0	11
Register windows	12	24	6	48
Total	97	154	127	234

Table 3-1: Breakup CAB-CAB latency (microseconds)

3.1. Analysis of CAB-CAB latency numbers

The datagram latency is the easiest to analyze (Figure 2-2). On transmit, one word is sent from an area in the applications stack, and the main cost is in the communication protocol processing. On receive, the datalink and datagram protocols are invoked through upcalls in the interrupt handler. They place the message in a mailbox and wake up the blocked application thread. Since that thread was the last one to be active, there is no thread switch overhead. The receiving thread reads the message from the mailbox and frees the buffer space.

The request-response protocol uses responses (requests) as acknowledgements of earlier requests (responses). As a result, the request-response protocol has to save messages (in a mailbox) before they are sent, and when receiving a response (request), it has to free the matching request (old response). This results in a higher mailbox overhead than in the datagram test. The request-response protocol also generates more window overflows and underflows since it has a higher call depth than the datagram protocol.

The RMP test has to explicitly acknowledge a packet before the receiving thread can run; this results in a high datalink overhead on receive. Between every send and receive the RMP test also has to handle the acknowledgement packet. This actually distorts the latency measurements because the incoming acknowledgement slows down the CAB, and the message arrives before the application thread blocks. This reduces the mailbox, register window and condition broadcast costs, compared with an isolated one-way communication operation.

While the Nectar-native protocols are handled inside the interrupt handling on receive, UDP protocol processing is done inside a protocol thread; IP processing is still done inside the interrupt handler. This difference shows up in Table 3-1 as a higher thread and condition overhead since first the protocol thread and then the receiving thread have to be woken up. Since the message is passed between the interrupt handler and the threads through mailboxes, there is also a higher mailbox cost.

In the remainder of this section, we look in more detail at the cost components of the communication latency.

3.2. Context switching

The overheads in Table 3-1 for interrupt handling, conditions, threads, and, indirectly, register windows are all associated with some form of context switching. They account for about 25% of the message latency.

3.2.1. Threads and context saving/restoring overhead

Table 3-2 breaks up the context switch overhead into time spent on thread management and on saving and restoring state. The thread-related overhead includes the cost of signaling and waiting on condition variables and the cost of thread switching. The latter cost is only incurred for UDP, because in the tests with the Nectar-native protocols, the thread that is woken up is always the one that blocked last. Saving and restoring state breaks up in saving and restoring global registers when entering and leaving an interrupt handler, and managing register windows: saving and restoring windows and checking and updating registers that control the windows. One window overflow/underflow costs 6 microseconds.

	Dgram	RR	RMP	UDP
Threads				
Conditions	10	10	0	21
Thread switch	0	0	0	11
Total threads	10	10	0	32
(% of latency)	(10.3%)	(6.5%)	(0%)	(13.7%)
Save/restore state				
Global state	3	3	3	3
Register windows	14.3	26.3	8.3	50.3
Total save/restore	17.3	29.3	11.3	53.3
(% of latency)	(17.8%)	(19.0%)	(8.9%)	(22.8%)
Total context switch	27.3	39.3	11.3	85.3
(% of latency)	(28.1%)	(25.5%)	(8.9%)	(36.5%)

Table 3-2: Context switch overhead in CAB-CAB latency (microseconds)

The results in Table 3-2 show that the overhead of the threads package (we ignore RMP since its results are artificially low) is relatively low (6.5% to 13.7%). As expected, UDP has the highest cost, since the separate protocol thread creates context switches. The cost of saving and restoring the state during context switches is higher than the thread overhead, and it roughly triples the cost of context switching. Almost all this cost is the result of register windows.

For the Nectar-native protocols, each node has to handle two threads of control throughout the test: the application thread and the receiving interrupt handler. Since the interrupt handler can share the register file with the interrupted thread, register windows theoretically allow packets to be received without having to save any registers. In practice windows do have to be saved, because the combined call depths of the two threads exceed the number of windows available on SPARC. For the request-response test for example, 64 registers have to be saved and restored. This number is higher than what would have to be saved on a processor with a flat register file, so the register windows do not reduce our interrupt overhead, but increase it.

For the UDP test, each node has to handle three threads of control: the sending thread, the receiving interrupt handler, and the receiving protocol thread. This organization results in two extra thread-thread switch, thus further increasing the register window overhead. As pointed out earlier, the UDP organization is more attractive, because it is easier to implement and maintain and does not mask interrupts for as long. Unfortunately, the thread implementation and, especially, register windows make this organization expensive. With a flat register file, the cost of doing protocol processing in a thread instead of an interrupt handler should drop to less than 30 microseconds.

3.2.2. Use of register windows

In our test programs, six windows are typically sufficient to hold the context of one thread of control and the number of windows in the SPARC implementation was probably chosen based on similar measurements. We incur a substantial overhead if we try to place two thread contexts in the register file. The SPARC architecture supports up to 32 windows, and we decided to change the CAB simulator to simulate a CAB with a SPARC processor with 13 (= 12 + 1) register windows, i.e. enough windows to hold two contexts. Table 3-3 shows the new context switch overheads. For the Nectar-native protocols the window saving and restoring disappears, as was to be expected, but for UDP, the overhead remains the same. The reason is that threads never occupy register windows simultaneously, so the number of window overflows and underflows remains the same, independent from the number of windows.

	Dgram	RR	RMP	UDP
Threads				
Conditions	10	10	0	21
Thread switch	0	0	0	11
Total threads	10	10	0	32
Save/restore state				
Global state	3	3	3	3
Register windows	2.3	2.3	2.3	50.3
Total save/restore	5.3	5.3	5.3	53.3

Table 3-3: Context switch overhead for CAB-CAB latency with 13 windows (microseconds)

The number of windows used by a thread of control is of course not a fundamental property. We could for example reduce the call depth of both threads of control in the datagram test so that their sum is six or less, thus eliminating register save and restore overhead. This is not a desirable optimization: procedure boundaries should not be based on the size of the register file. Moreover, this optimization is not done automatically by the compiler, and is not visible to the programmer. Changes to the code can easily add a level to the call depth, thus undoing the optimization.

An interesting observation is that from the point of view of register windows, threads usually blocks at the ‘‘wrong’’ time: when the call stack is relatively deep and several windows are in use. The reason is that communication operations are often implemented as several layers of abstraction. The left column in Figure 3-1 shows the layers for blocking send and receive operations. Higher levels of abstraction that hide the send and receive operations from the application can further increase the call depth. Each layer typically adds at least one level to the call tree.

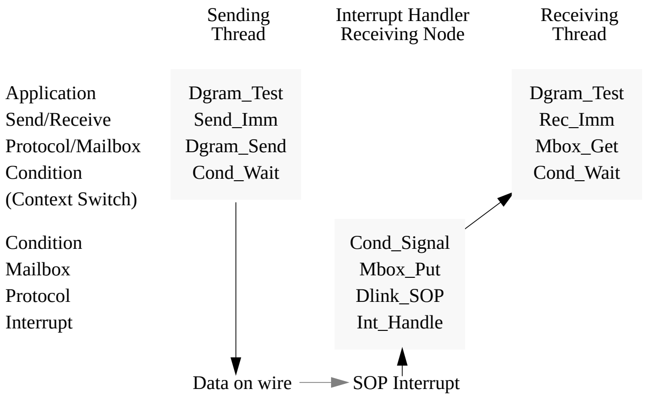

Figure 3-1: Layers of abstraction for datagram

The right side of Figure 3-1 shows the routines that are executed for the CAB-CAB datagram test. Each layer is implemented as a single procedure call. The bottom of Figure 3-1 shows the layers for the start-of-packet trap handler. The sum of the call depths of the blocked receiving thread (4) and of the trap handler (4) exceeds the number of usable windows (6). Figure 3-2 shows the layers for the more complicated UDP test.

3.3. Buffer management

Mailbox operations account for about 20-25% of the CAB-CAB latency. Most of this cost is on the receive side. All mailbox operations in the tests follow the ‘‘fast path’’ through the code. Each mailbox has space to cache one small empty buffer (508 bytes). When a small packet is put in the mailbox, this buffer is used if it is available, and similarly, when a small packet is freed, the buffer is cached if the cache is empty. The cached buffer optimizes the case when performance is most critical: an application

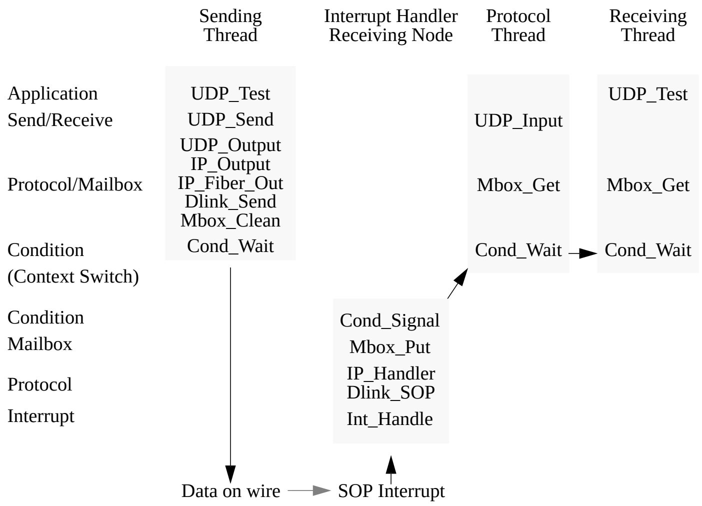

Figure 3-2: Layers of abstraction for UDP

has consumed a message and is waiting for the next message. If it is not possible to use the cached empty buffer, the mailbox module allocates space for the message using the generic memory allocation module. This roughly doubles the cost of placing a message in a mailbox.

Placing a message in a mailbox and having the application read it and free the buffer costs about 25 microseconds. This is relatively expensive, but all the operations that are performed (checking the address, allocating and freeing space, queueing and dequeueing the message, ...) are necessary. They are costly because they are memory intensive. One non-essential feature that adds overhead to the fast path through the mailbox module is the ability to reduce the size of messages in mailboxes. This increases the size of the message descriptor and adds overhead, even if the feature is not used. However, this feature makes it easy to remove transport protocol headers, and dropping it would increase the protocol cost. The fact that messages in mailboxes can be consumed by both host processes and CAB threads also adds a small cost. Specifically, the CAB has to signal both a host and a thread condition variable when it places a message in a mailbox.

The main reason why buffer management is so expensive, especially for longer messages, is that messages have to be placed in memory contiguously since they can be read directly by applications. Traditional buffer management strategies such as placing messages in a sequence of fixed size blocks do not work since the application would not see a contiguous message. The only way to speed up buffer management is to eliminate some flexibility. For example, limiting the message size or restricting the order in which messages can be freed would simplify the mailbox and memory management.

The primary use of mailboxes is to hold packets that were sent over Nectar, but mailboxes are used in other ways in the test programs. First, they function as retransmission buffers in the RR and UDP tests and second, messages are passed between the IP and UDP protocol layers through a mailbox. In both of these cases, some of the functionality of the mailboxes is not used, and the overhead could be reduced by using a simpler, special-purpose mechanism.

3.4. Communication protocols

The communication protocols account for between 25% and 40% of the CAB-CAB latency. For the Nectar-native protocol tests, this overhead is dominated by the datalink protocol. As discussed elsewhere [11], the datalink is mostly done in software for flexibility reasons, and most of it could be handled in hardware, thus reducing the protocol overhead.

The transport protocol cost ranges from less than 10 microseconds (datagram), to 25-30 microseconds (request-response and RMP), and 50 microseconds (UDP). The overhead for the reliable Nectar-native protocols is of the order of 200 instructions, i.e. similar to that reported in the literature for optimized standard protocols [7]. UDP is a lot more expensive, mainly because it has not been optimized. It is also more general than the Nectar-native protocols, since UDP packets can travel outside Nectar.

3.5. CAB-CAB latency summary

Table 3-4 summarizes the results of the CAB-CAB latency analysis. We see that transport protocols are responsible for less than 25% of the latency and that both context switching and buffer management are more expensive. The table also shows clearly that a thread-based protocol implementation is substantially slower than an implementation that does protocol processing in an upcall, at least for a processor that uses register windows. Using a flat register file would reduce the difference between the two implementations by about 50%.

	Dgram	RR	RMP	UDP
Datalink prot	32 (33%)	36 (23%)	65 (44%)	37 (16%)
Transport prot	6 (6%)	25 (16%)	31 (24%)	50 (21%)
Mailbox	30 (31%)	52 (34%)	27 (21%)	60 (26%)
Context sw	29 (30%)	41 (27%)	13 (10%)	87 (37%)
Total	97	154	127	234

Table 3-4: Breakup CAB-CAB latency (microseconds)

If we would reimplement the CAB architecture today as a computing node in a multicomputer, we would be able to reduce the latency in several ways. First, implementing the datalink in hardware would reduce the datalink overhead. Second, transport protocol and buffer management would benefit from more efficient implementations and faster hardware. The context switch overhead is the most troublesome. Although this cost can be reduced by using a flat register file, adding caches and extending the runtime system (e.g. add virtual memory) would increase the cost of context switching. Context switching has traditionally also benefited little from increased CPU speed [2].

4. Host-host breakup

A comparison of Figures 2-2 and 2-3 shows that the behavior of the threads on the CAB is the same for the CAB-CAB and host-host roundtrip tests. As a result, we can use the CAB-CAB measurements to analyze the CAB component of the host-host roundtrip times. The host-CAB interaction overhead was measured separately on the hardware using a microsecond clock on the CAB. The results are shown in Table 4-1. The bottom line shows that our estimates are very close to the measured latency.

	Dgram	RR	RMP	UDP
Host to CAB	52	52	52	52
Send				
Interrupt handling	7	7	7	7
Conditions	10	10	10	10
Datalink protocol	14	12	12	14
Transport protocol	6	7	22	23
Mbox	8	8	8	8
Register windows	0	0	0	24
Thread switch	0	0	0	6
Receive				
Interrupt handling	7	7	7	7
Conditions	0	0	0	10
Datalink protocol	18	24	24	23
Transport protocol	0	18	7	27
Mbox put	18	25	18	38
Register windows	6	18	12	24
Thread switch	0	0	0	5
CAB to Host	26	26	26	26
Total estimate	172	214	205	304
Measured latency	169	225	213	308

Table 4-1: Estimated breakup host-host latency (microseconds)

In comparison with the CAB-CAB tests (Table 3-1), the number of register underflows and overflows on the CAB on receive is reduced by one for most tests, because the protocol threads are one call less deep than the user threads. The RMP protocol processing overhead on receive is substantially smaller than in the CAB-CAB test because the application on the host can execute while the packets is being acknowledged. The anomaly in the RMP CAB-CAB measurements caused by the acknowledgement does not occur in the host-host test.

4.1. Analysis of host-host latency

The main difference between the host-host and CAB-CAB latency is the addition of the host-CAB interaction. The host sends a message by placing it in a mailbox in CAB memory and receives a message by reading it from a mailbox in CAB memory. Both operations are expensive because they have to be performed across the VME bus. Host accesses to CAB memory take about 1.1 microseconds, and two thirds of the host time is spent reading and writing across the VME bus. Note that the time to read and write individual words across more recent IO busses such as Turbo Channel [19] is still about 1 microseconds, so single word accesses across IO busses are not getting much faster.

Sending and receiving messages requires so many host accesses to CAB memory for a number of reasons. First, transport protocols are invoked ‘‘indirectly’’ through mailboxes, a general-purpose mechanism for host-CAB communication. An alternative is to give transport protocols a privileged status on the CAB, and to invoke them through a special-purpose mechanism. Requests for the transport protocols could for example be placed in a special-purpose queue, that is always checked by the CAB when it is interrupted by the host. This would eliminate several operations, such as various mailbox checks and the building of a request for the CAB interrupt handler to wake up the protocol thread. This optimization would reduce the host-to-CAB overhead, but the cost would be a more complex host-CAB interface since there would be two mechanisms for invoking services on the CAB.

A second reason for the large number of host accesses to CAB memory is that buffer management is done entirely in software. The data structures used for the management of memory and mailboxes are located in CAB memory and queueing or dequeueing a message or empty buffer requires a certain number of memory accesses. Hardware support for message queues would reduce this cost: enqueue and dequeue operations could be performed with a single access across the VME bus. The flexible CAB runtime system also contributes to the high cost of host operations on mailboxes. The CAB is a computer system in its own right, and can for example destroy mailboxes or alter their properties. As a result, the host has to check for every operation on a mailbox whether the status of the mailbox has changed since the last operation; this is not complicated but it does requires CAB accesses. The host and CAB currently form an asymmetric multiprocessor. If the CAB were purely a slave, the mailbox structure could be simplified.

4.2. Cost of flexibility

The flexible runtime system on the CAB increases the host-host latency in several ways, and it is interesting to consider how the latency would change if we used the CAB as a hardwired protocol engine. This CAB ''controller'' could for example continuously poll the interrupt lines to detect network events, send requests from the host, and timeout events for the communication protocols. The available CAB data memory would be organized as two pools of fixed-sized buffers, one for outgoing and one for incoming packets; note that this places an upperbound on the packet size. Messages would be received in simplified mailboxes, i.e. FIFOs with a network address. Table 4-2 shows rough estimates for the host-host latency in a Nectar system with hardwired protocol engines, assuming the same hardware and protocol processing costs.

	Dgram	RR	RMP	UDP
Host to CAB	20	20	20	20
Send				
Interrupt handling	2	2	2	2
Datalink Protocol	14	12	12	14
Transport Protocol	6	7	22	23
Invoke transport	3	3	3	3
Receive				
Interrupt handling	2	2	2	2
Mailbox put	10	10	10	10
Datalink Protocol	18	24	24	23
Transport Protocol	0	18	7	27
CAB to Host	20	20	20	20
Total dumb CAB	95	118	131	144
Total smart CAB	172	214	205	304
Speedup	45%	45%	35%	53%

Table 4-2: Estimated host-host latency using a hardwired protocol engine (microseconds)

Table 4-2 show that the latency would be reduced by 35%-53%. Most of the saving is the result of the elimination of the saving and restoring of state on the CAB, and of the simplification of the host-CAB interface. The highest speedup is for UDP, which was making most use of the CAB runtime system. The cost for host-CAB interaction remains relatively high because of the high cost of VME accesses. On transmit, the cost includes taking a block from the free queue in CAB memory, writing a header and a data word, and queueing the packet for transmission. Locks are required at least for the free queue since it is shared by all host processes. A receive operation consists of dequeueing a message, reading the data, and enqueue the buffer on the free queue, i.e. the fast path in the original mailbox implementation.

The results in Table 4-2, as all host-host latency results reported, are for the case where the message arrives shortly after the application on the host tries to read it, i.e. the ideal case. Once the host process has been put to sleep, the host-host latency increases. The CAB-CAB latencies reported in the previous section do not have that problem.

4.3. Host-host latency summary

The host-host latency is significantly higher than the CAB-CAB latency because of the high cost of host-CAB interactions. The host-host communication latency can be reduced in several ways. First, using a special-purpose mechanism to invoke transport protocols could reduce the cost of sending messages. Second, adding hardware support for queue management can minimize the impact of the high overhead of accessing individual words across the IO bus. Finally, making the CAB a hardwired protocol engine would significantly reduce the overhead on the CAB and would also reduce the complexity and cost of the host-CAB interface. We would of course also lose the ability to move application code to the CAB.

5. Other performance measures

The estimated host-host latency using a hardware protocol engine are similar to the CAB-CAB performance numbers. One could conclude that it would be better to use the CAB as a hardwired protocol engine, instead of implementing a flexible CAB runtime that supports applications. This conclusion is misleading and the reason is that we take into account only the best case latency for short packets as a performance measure. If we consider other performance measures, such as throughput and the latency when the receiving host process has been put to sleep, the host-host numbers are worse than the CAB-CAB numbers.

Besides latency, two other important performance measures are throughput and the overhead associated with sending and receiving messages, i.e. how many cycles does an application lose when it communicates. The message overhead for Nectar is low compared with that of other networks: it consists only of the cost of placing a message in or retrieving a message from a mailbox in CAB memory (Table 4-1). All other operations associated with communication are performed on the CAB, in parallel with application processing on the host. Inexpensive communication is important for many applications.

The host-host throughput with 8KByte packets [8] is 3 MByte per second; it is limited by the VME bus. The CAB-CAB throughput with 8KByte packets is about 11 MByte per second. The bottleneck created by the VME bus was one of the main motivations for supporting applications on the CAB, since this gives applications access to the network without having to cross the VME bus. For example, we have observed cases where a server on the CAB handled 10000 request per second; this performance would not have been possible on the host.

We have looked at several host-network interfaces in this paper. First we considered the CAB architectures. The network interface does not provide any support for protocol processing, but it is closely integrated with the CPU-memory system: it has its own fast path into memory (Figure 2-1) and CPU accesses to the network device are as fast as memory accesses, so the CPU can control the network interface very quickly. This network interface offers the best results for two of the three performance measures: latency for short packets and throughput.

We then considered the interface of Nectar hosts: the hosts communicate over Nectar using an intelligent network interface that handles protocol processing. This interface has a higher latency for short packets, even if we give up the flexible runtime system on the CAB (Table 4-2) and it also has a lower throughput because of the VME bottleneck. Sending and receiving (short) messages is however less expensive on Nectar hosts than on CABs, since most of the communication overhead (protocol processing and buffer management) are performed in parallel with application processing.

It should not be a surprise that a tight coupling between the CPU and network interface, as we find on the CAB, allows us to get good latency and throughput. This is for example how dedicated multicomputers achieve good performance. Workstations consist of a core that includes the CPU and the memory system, and all other components are relegated to an IO bus. This organization makes building high-performance network interfaces much more challenging. For example, host accesses across the IO bus are expensive, and if DMA is used, there is a lot of interaction between accesses to host memory by the network interface and the CPU, in part because the cache and memory have to be kept consistent in software.

6. Earlier work

Other work analyzing the overhead associated with network communication includes TCP and UDP communication between two Vax 11/750 machines [5], the performance of the Firefly RPC mechanism [16], the performance of the remote accesses over Ethernet [18], the x-kernel roundtrip time [10], and the performance of the Amoeba RPC [14]. Since all these systems are organized differently and often use different protocols, it is hard to compare the results, but we can make some general observations.

First, all measurements seem to agree that protocol processing is only one part of the overhead; other costs such as interrupt handling, dealing with the hardware controller (included in the datalink processing overhead in our case), buffer management and thread or process scheduling are more significant, certainly for short packets. Second, the cost of buffer management for Nectar is typically higher than for the other implementations (if reported), with the exception for the TCP/UDP measurements reported in [5]. The reasons are the flexibility of the mailbox module, and the requirement that messages have to be contiguous in memory.

7. Conclusion

We presented an analysis of host-host and CAB-CAB communication latency in Nectar. The CAB-CAB measurements give some insights in communication latency between nodes using light-weight runtime systems. We found that four sources of overhead are each responsible for about one quarter of the latency: context switching, buffer management and the datalink and transport protocols. A significant component of the context switch overhead is the saving and restoring of register windows on the SPARC CPU. Faster hardware and more optimized implementations can probably reduce the cost of the last three components. The context switch overhead is not likely to drop significantly and will be significant bottleneck in low latency communication.

Host-CAB interactions form a significant component of the Nectar host-host latency. The primary reason is that single word accesses across the IO bus (VME bus for Nectar) are expensive. Provide queueing hardware would speed up the host-CAB interaction, since messages and empty buffers can be exchanged using a single transfer over the bus. The best solution would of course be to integrate the network interface closely with the CPU and memory system of the workstation, as is done on the Nectar CAB.

Acknowledgements

I would like to thank all the people who contributed to the Nectar project, especially the people involved in the development of the communication software: Eric Cooper, Robert Sansom and Brian Zill. I would also like to thank I-Chen Wu, Fred Christianson and Denise Ombres for making some of the measurements.

References

1. Special Report. "Gigabit Network Testbeds". *IEEE Computer 23*, 9 (September 1990), 77-80.

2. Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, April, 1991, pp. 108-121.

3. Emmanuel Arnould, Francois Bitz, Eric Cooper, H. T. Kung, Robert Sansom and Peter Steenkiste. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ACM/IEEE, Boston, April, 1989, pp. 205-216.

4. D. H. Bailey, E. Barszcz, R. A. Fatoohi, H. D. Simon, and S. Weeratunga. Performance Results on the Intel Touchstone Gamma Prototype. Proceedings of the Fifth Distributed Memory Computing Conference, IEEE, April, 1990, pp. 1236-1245.

5. Luis-Felipe Cabrera, Edward Hunter, Michael J. Karels, and David A. Mosher. "User-Process Communication Performance in Networks of Computers". *IEEE Transactions on Software Engineering 14*, 1 (January 1988), 38-53.

6. David D. Clark. The Structuring of Systems Using Upcalls. Proceedings of the Tenth Symposium on Operating System Principles, ACM, December, 1985, pp. 171-180.

7. David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. "An Analysis of TCP Processing Overhead". *IEEE Communications Magazine 27*, 6 (June 1989), 23-29.

8. Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol Implementation on the Nectar Communication Processor. Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols, ACM, Philadelphia, September, 1990, pp. 135-143.

9. Eric C. Cooper and Richard P. Draves. C Threads. Tech. Rept. CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June, 1988.

10. Norman C. Hutchinson and Larry L. Peterson. Implementing Protocols in the x-kernel. Tech. Rept. 89-1, University of Arizona, January, 1989.

11. H.T. Kung, Robert Sansom, Steven Schlick, Peter Steenkiste, Matthieu Arnould, Francois J. Bitz, Fred Christianson, Eric C. Cooper, Onat Menzilcioglu, Denise Ombres, and Brian Zill. Network-Based Multicomputers: An Emerging Parallel Architecture. Proceedings of Supercomputing '91, IEEE, Albequerque, November, 1991, pp. 664-673.

12. H.T. Kung, Peter Steenkiste, Marco Gubitoso, and Manpreet Khaira. Parallelizing a New Class of Large Applications over High-Speed Networks. Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, April, 1991, pp. 167-177.

13. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

14. Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. "Amoeba: A Distributed Operating System for the 1990s". *IEEE Computer 23*, 5 (May 1990), 44-53.

15. Larry Peterson, Norman Hutchinson, Sean O'Malley, and Herman Rao. "The x-kernel: A Platform for Accessing Internet Resources". *IEEE Computer 23*, 5 (May 1990), 23-34.

16. Michael Schroeder and Michael Burrows. "Performance of Firefly RPC". *ACM Transactions on Computer Systems 8*, 1 (February 1990), 1-17.

17. *MB86900 RISC Processor - Architecture Manual.* Fujitsu, 1987.

18. Alfred Spector. Communication Support in Operating Systems for Distributed Transactions. Tech. Rept. CMU-CS-86-165, Computer Science Department, Carnegie Mellon University, November, 1986.

19. *TURBOchannel Overview.* Digital Equipment Corporation, 1990.