Protocol Implementation on the Nectar Communication Processor

Eric C. Cooper, Peter A. Steenkiste, Robert D. Sansom, and Brian D. Zill

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

We have built a high-speed local-area network called Nectar that uses programmable communication processors as host interfaces. In contrast to most protocol engines, our communication processors have a flexible runtime system that supports multiple transport protocols as well as application-specific activities. In particular, we have implemented the TCP/IP protocol suite and Nectar-specific communication protocols on the communication processor. The Nectar network currently has 25 hosts and has been in use for over a year.

The flexibility of our communication processor design does not compromise its performance. The latency of a remote procedure call between application tasks executing on two Nectar hosts is less than 500 μsec. The same tasks can obtain a throughput of 28 Mbit/sec using either TCP/IP or Nectar-specific transport protocols. This throughput is limited by the VME bus that connects a host and its communication processor. Application tasks executing on two communication processors can obtain 90 Mbit/sec of the possible 100 Mbit/sec physical bandwidth using Nectar-specific transport protocols.

The research was supported in part by the Defense Advanced Research Projects Agency (DOD) and monitored by DARPA/CMO under Contract MDA972-90-C-0035. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the U.S. Government.

1 Introduction

The protocols used by hosts for network communication can be executed on the host processors or offloaded to separate communication processors. In Nectar, a high-speed local-area network, we have taken the latter approach. By offloading transport protocol processing from the host to the communication processor, we reduce the burden on the host. Executing protocols on a communication processor is also attractive if the host is unsuited to protocol processing, as in the case of specialized architectures, or if the host operating system cannot easily be modified, as in the case of supercomputers.

Unlike traditional network front-end processors, the Nectar communication processor has a general-purpose CPU and a flexible runtime system that support both transport protocol processing and application-specific tasks. Protocol implementations on the communication processor can be added or optimized with no change to the host system software; this is particularly advantageous in environments with heterogeneous hardware and operating systems. Application-specific communication tasks can be developed for either the host or the communication processor using the *Nectarine* programming interface.

The interface between the communication processor and user processes on the host is based on shared memory. The buffer memory of the communication processor is directly accessible to user processes. No system calls or user-to-kernel copy operations are required to send and receive messages. As a result, host processes can communicate with lower latency (by a factor of 5) than would be possible using the UNIX socket interface of the host operating system [13].

Related work on host interfaces for high-speed networks includes the VMP Network Adapter Board [10] and the Protocol Engine design [4]. In these approaches, processing of specific transport protocols is offloaded to the network interface, but there is no provision for the execution of application-specific tasks or multiple transport protocols.

The Nectar runtime system is similar in structure to other operating systems designed specifically to support network protocols, such as Swift [7] and the *x*-kernel [12]. However the Nectar system is distinguished by its emphasis on the interface between the communication processor and the host.

The Nectar communication processor together with its host can be viewed as a (heterogeneous) shared-memory multiprocessor. Dedicating one processor of a multiprocessor host to communication tasks can achieve some of the benefits of the Nectar approach, but this constrains the choice of host operating system and hardware. In contrast, the Nectar communication processor has been used with a variety of hosts and host operating systems.

In this paper we describe and evaluate our approach to building a communication processor. We first give an overview of the Nectar hardware (Section 2). We then describe the design of the runtime system on the communication processor and the interactions between the runtime system and host processes. As an example of the use of the runtime system, we discuss our implementation of TCP/IP in Section 4. In Section 5, we discuss how the flexibility of the Nectar design allows different levels of communication functions to be offloaded from the host to the communication processor. A performance evaluation of the Nectar system is presented in Section 6.

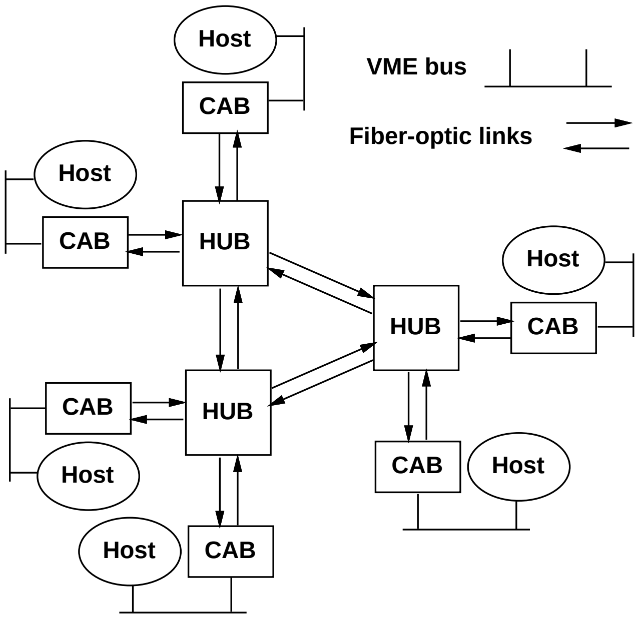

Figure 1: Nectar system overview

2 The Nectar System

The Nectar system consists of a set of host computers connected in an arbitrary mesh via crossbar switches called *HUBs* (Figure 1). Each host uses a communication processor, called a *CAB* (*Communication Accelerator Board*), as its interface to the Nectar network. More details about the Nectar architecture can be found in an earlier paper [2].

2.1 HUB Overview

The Nectar network is built from fiber-optic links and one or more HUBs. A HUB consists of a crossbar switch, a set of I/O ports, and a controller. The controller implements commands that the CABs use to set up both packet-switching and circuit-switching connections over the network.

Large Nectar systems are built using multiple HUBs. In such systems, some of the HUB I/O ports are used to connect together HUBs. The CABs use source routing to send a message through the network. The HUB command set includes support for multi-hop connections and low-level flow control.

In the current Nectar system, the fiber-optic lines operate at 100 Mbit/sec and the HUBs are 16×16 crossbars. The hardware latency to set up a connection and transfer the first byte of a packet through a single HUB is 700 nanoseconds.

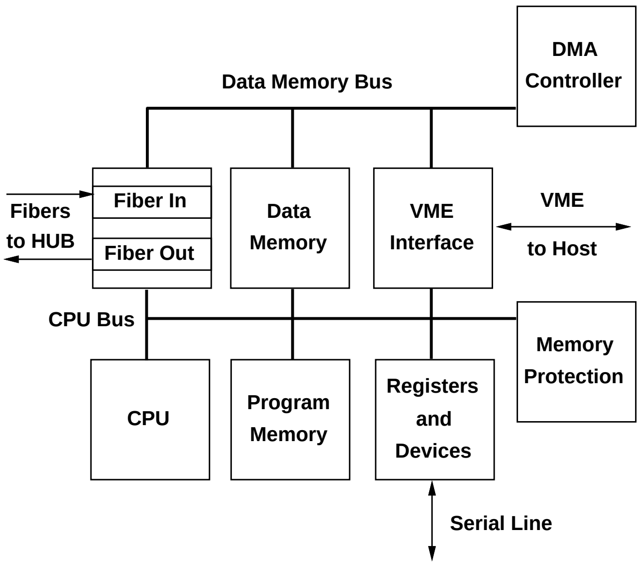

Figure 2: CAB block diagram

2.2 CAB Overview

A block diagram of the CAB is shown in Figure 2. The heart of the CAB is a general-purpose RISC CPU. Two optical fibers—one for each direction—connect the CAB to an I/O port on the HUB. The fibers are connected to FIFOs for temporary buffering of network data. Cyclic Redundancy Checksums for incoming and outgoing data are computed by hardware. The CAB communicates with the host through a VME interface, a common backplane in our environment.

The CAB includes a hardware DMA controller that can manage simultaneous data transfers between the incoming and outgoing fibers and CAB memory, as well as between VME and CAB memory, leaving the CAB CPU free for protocol and application processing. The DMA controller also handles low-level flow control for network communication: it waits for data to arrive if the input FIFO is empty, or for data to drain if the output FIFO is full.

To provide the necessary memory bandwidth, the CAB memory is split into two regions: one intended for use as program memory, the other as data memory. DMA transfers are supported for data memory only; transfers to and from program memory must be performed by the CPU. The memory architecture is thus optimized for the expected usage pattern, although still allowing code to be executed from data memory or packets to be sent from program memory.

Memory protection hardware on the CAB allows access permissions to be associated with each 1 Kbyte page. Multiple protection domains are provided, each with its own set of access permissions. Changing the protection domain is accomplished by reloading a single register.

The current CAB implementation uses a SPARC processor running at 16.5 MHz. The program memory region contains 128 Kbytes of PROM and 512 Kbytes of RAM. The data memory region contains 1 Mbyte of RAM. Both memories are implemented using 35 nsec static RAM.

3 Runtime System

The CAB runtime system must support concurrent activities that include network interrupts, transport-protocol processing, and application-specific computation. A lightweight interface between the host and the CAB is also essential; expensive host-CAB synchronization, data copying, and system calls must be avoided.

Figure 3 shows the structure of the Nectar software on the host and CAB. The basic CAB runtime system provides support for multiprogramming (the threads package) and for buffering and synchronization (the mailbox and sync modules). Transport protocols (described in Section 4) are implemented on the CAB using these facilities. The Nectarine layer provides a consistent interface for applications on both the CAB and the host. The CAB device driver in the host operating system allows host processes to map CAB memory into their address spaces.

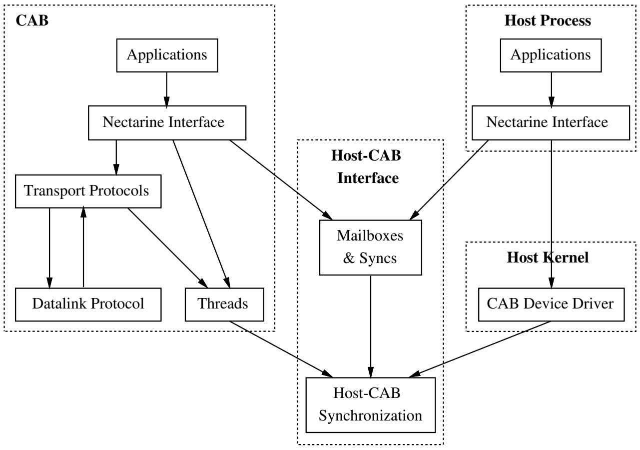

Figure 3: Nectar software architecture

3.1 Threads, Interrupts, and Upcalls

Previous protocol implementations have demonstrated that multiple threads are useful, but multiple address spaces are unnecessary [7, 11, 12]. Since we expected most of the activities on the CAB to be protocol-related, we designed the CAB to provide a single physical address space, and the runtime system to support a single address space shared by multiple threads. The runtime system can use the multiple protection domains described in Section 2 to provide firewalls around application tasks if desired.

The threads package for the CAB was derived from the Mach C Threads package [8]. It provides forking and joining of threads, mutual exclusion using locks, and synchronization by means of condition variables. Context switch time is determined by the cost of saving and restoring the SPARC register windows; 20 μsec is typical in the current implementation.

System threads (such as those implementing network protocols) are typically driven by events such as a packet arriving or a condition being signaled; after a brief burst of processing, they relinquish the processor by waiting for the next event. We make no such assumptions about application threads: they may perform long computations with few synchronization points, or they may get stuck in infinite loops. Preemption of application threads is therefore necessary. The current scheduler uses a preemptive, priority-based scheme, with system threads running at a higher priority than application threads.

Before we implemented preemptive scheduling of threads, *upcalls* [7] from interrupt handlers were the only way to provide sufficiently fast response to external events. For example, because of the speed at which an incoming packet fills the CAB input FIFO, a start-of-packet interrupt must be handled within a few tens of microseconds. Waking up another thread has unacceptably long response time—the context switch would not occur until the currently running thread reached a

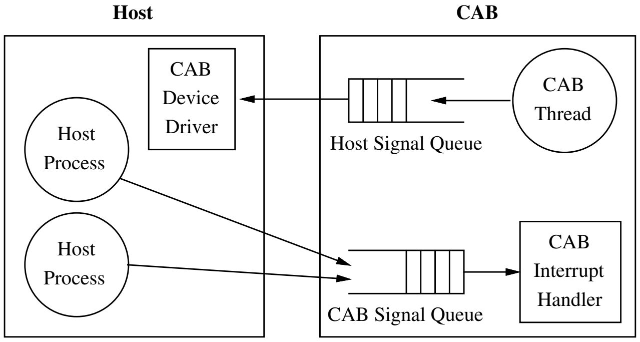

Figure 4: Host-CAB signaling

synchronization point and relinquished the processor.

Using upcalls from interrupt level means that data structures must be shared between threads and interrupt handlers, resulting in critical sections that must be protected by appropriate masking of interrupts. Disabling interrupts is less elegant than protecting critical sections by means of module-specific mutual exclusion locks because it violates modularity. The implementor of an abstraction must know whether its callers are threads or interrupt handlers so that interrupts can be masked appropriately.

With preemption, a context switch occurs as soon as a higher-priority thread is awakened. We therefore plan to revisit our decision to perform significant amounts of protocol processing at interrupt time. We will experiment with moving portions of it into high-priority threads. Although this will introduce additional context switching, the CAB will spend less time with interrupts disabled, so overall performance is likely to improve.

The response time could also be improved by using the SPARC's interrupt priority scheme to implement nested interrupts. Although appropriate use of nested interrupts could further reduce latency, the cost would be greater complexity and lack of modularity in the code because the implementor of an abstraction would have to be aware of the possible interrupt priority levels of users of the abstraction.

3.2 Host-CAB Signaling

Host processes and CAB threads interact using shared data structures that are mapped into the address spaces of the host processes. To manipulate data structures in CAB memory, a host process must be able to do the following:

- Map CAB memory into its address space and translate between CAB physical addresses and host virtual addresses.

- Wait for synchronization events on the CAB using either polling or blocking.

- Notify CAB threads and host processes that an event has occurred.

The CAB device driver in the host operating system enables host processes to map CAB memory into their address spaces (by using the `mmap` system call). This mapping is done as part of program initialization.

Host condition variables are used for host-CAB synchronization. Host condition variables are similar to the condition variables in the threads package on the CAB, except that the waiting entities are host processes instead of CAB threads. Host condition variables are located in CAB memory where they can be accessed by both CAB threads and host processes.

`Signal` and `Wait` are the main operations on host conditions. `Signal` increments a *poll value* in the host condition. `Wait` repeatedly tests the poll value, and returns when the poll value changes. Both CAB threads and host processes can signal a host condition. Using polling, host processes can wait for host conditions without incurring the overhead of a system call.

In many situations, for example a server process waiting for a request, polling is inappropriate because it wastes host CPU cycles. Thus we also allow host processes to `Wait` for host conditions without polling, by calling the CAB device driver. The CAB driver records that the process is interested in the specified host condition and puts the process to sleep. When a host condition variable is signaled, its address is placed in the *host signal queue* (Figure 4), and the host is interrupted. The CAB driver handles the interrupt and uses the information in the queue to wake up the processes that are waiting for the host condition.

The host signal queue has fixed-size elements that consist of an opcode and a parameter. This queue can also be used by the CAB for other kinds of requests to the host, such as invocation of host I/O and debugging facilities.

Host processes wake up CAB threads by placing a request in the *CAB signal queue* (Figure 4) and interrupting the CAB. As with the host signal queue, the CAB signal queue is also used to pass other types of requests to the CAB.

The CAB signaling mechanism is extended into a simple host-to-CAB RPC facility by allowing the CAB to return a result to the host. The *sync* abstraction described in Section 3.4 provides the necessary synchronization and transfer of data.

3.3 Mailboxes

A mailbox is a queue of messages with a network-wide address. The buffer space used for the messages associated with a mailbox is allocated in CAB memory. By mapping CAB memory into their address spaces, host processes can build and consume messages in place.

Mailboxes also provide synchronization between readers and writers. A host process or a CAB thread blocks when it tries to read a message from an empty mailbox; it resumes when a message has been placed in the mailbox, typically by a transport protocol on the CAB.

These features make mailboxes attractive for communication between the host and the CAB. A host process can invoke a service on the CAB by placing a request in a server mailbox; this wakes up the server which processes the request and places the result in a reply mailbox, where it can be read by the host process. Similarly, a CAB thread can invoke a service on the host by placing a request in a mailbox that is read by a host process.

Network-wide addressing of mailboxes enables host processes or CAB threads to send messages to remote mailboxes via transport protocols. In this way, remote services can be invoked from anywhere in the Nectar network.

The Mailbox Interface

A two-phase scheme is used for both reading and writing messages in mailboxes. This allows messages to be produced or consumed in place without further copying. Figure 5 depicts the state transitions that a message undergoes as a result of the mailbox operations.

To write a message, a program first calls `Begin_Put`, specifying the mailbox and the size of the message. This returns a pointer to a newly allocated data area of the required size. The writer can now fill in the contents of the message; space for additional messages may be reserved in the meantime using additional `Begin_Put` calls. When the program has finished writing the message, it uses `End_Put` to make the message available to readers.

A reader calls `Begin_Get` to obtain a pointer to the next available message, allowing the data to be read in place. When the reader is finished with the data, `End_Get` releases the storage associated with it. Multiple threads can use these operations to process concurrently the messages arriving at a single mailbox.

Some applications (such as IP, described in Section 4.1) need the ability to move a message from one mailbox to another. The operations described so far would allow this functionality, but at the cost of copying data between the different data areas associated with the two mailboxes. To avoid this overhead, we introduced an `Enqueue` operation that moves the message without copying the data. We also provided operations to "adjust" the size of messages in place, effectively removing a prefix or suffix of the message without doing any copying.

Both `Begin_Put` and `Begin_Get` block if no space or message is available. The calling thread is rescheduled when space becomes available or a message arrives. Interrupt handlers use non-blocking versions of these calls.

The mailbox interface also allows a *reader upcall* to be attached to a mailbox. The upcall is invoked as a side effect of the `End_Put` operation. This flexibility allows us to trade the concurrency of multiple threads against the overhead of context switching. For example, if a pair of threads uses a mailbox in a client-server style, the body of the server thread can instead be attached to the mailbox as a reader upcall; this effectively converts a cross-thread procedure call into a local one.

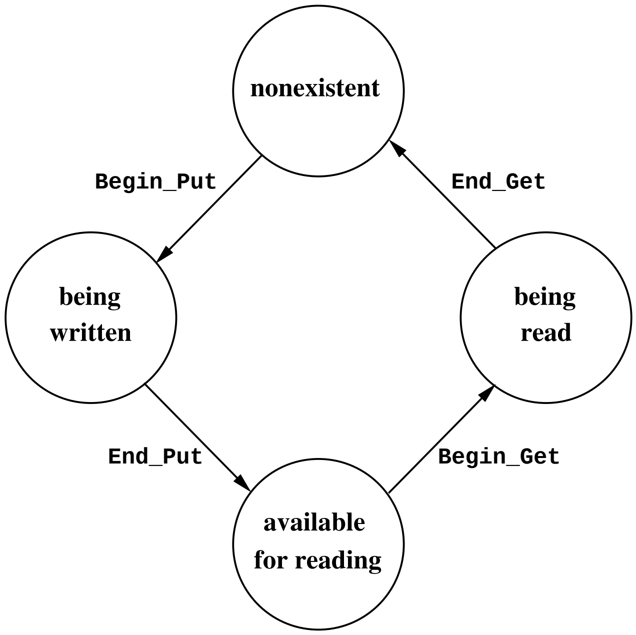

Figure 5: Mailbox operations and message states

Implementation of Mailboxes

Mailboxes are implemented as queues of messages waiting to be read; buffer space for messages is allocated from a common heap. Allocating buffers from the heap provides better utilization of the CAB data memory since it is shared among all mailboxes on the CAB. As an optimization, each mailbox caches a small buffer; this avoids the cost of heap allocation and deallocation when sending small messages. The queue representation also allows us to implement the `Enqueue` operation by simply moving pointers.

Mailbox operations from the host were initially implemented using the simple host-to-CAB RPC mechanism described in Section 3.2. We also implemented a shared memory version in which mailbox data structures are updated directly from the host. Since the reader and writer data structures are separate, mutual exclusion between CAB threads and host processes can be avoided as long as the readers either all reside on the CAB or all reside on the host, and the same for writers. This is certainly true in the common case of a single reader and a single writer, and also, more generally, for client-server interfaces across the host-CAB boundary.

In return for the restrictions on placement of readers and writers, the shared memory implementation provides about a factor of two improvement over the RPC-based implementation for Sun 4 hosts. We have configured the runtime system so that both implementations coexist, and the appropriate implementation can be selected dynamically on a per-mailbox basis.

3.4 Lightweight Synchronization

Synchronization between two threads or processes does not always need the full generality of mailboxes. For example, returning a status value from a transport protocol on the CAB to a sender on the host could be done using a mailbox, but all that is really needed is a condition variable and a shared word for the value. *Syncs* allow a user to return a one-word value to an asynchronous reader efficiently; they are similar to Reppy's *events* [14].

The Sync Interface

The operations on syncs are `Alloc`, `Write`, `Read`, and `Cancel`. `Alloc` allocates a new sync. `Write` places a one-word value in the sync data structure, and marks the sync as written. `Read` blocks until a sync has been written, then frees the sync and returns its value. Alternatively, the reader can use `Cancel` to indicate that it is longer interested in the sync. `Cancel` frees the sync if it has been written; otherwise `Cancel` just marks the sync as canceled, leaving it to be freed as part of a subsequent `Write`.

Implementation of Syncs

Host processes and CAB threads allocate syncs in CAB memory; conflicts are avoided by using two separate pools of syncs. Since there is only one reader, reading a sync does not require any locking. Writing a sync does require a critical section: checking whether the sync has already been canceled and marking the sync as written must be done atomically. On the CAB this is done by masking interrupts. A host process offloads the execution of `Write` to the CAB using the CAB signaling mechanism. `Cancel` is implemented similarly.

3.5 The Application Interface: Nectarine

Most of the current Nectar applications are written using *Nectarine*, the Nectar Interface. Nectarine is implemented as a library linked into an application's address space. It provides applications with a procedural interface to the Nectar communication protocols and direct access to mailboxes in CAB memory. It also allows applications to create mailboxes and tasks on other hosts or CABs. Nectarine simplifies the task of writing Nectar applications by hiding the details of the host-CAB interface and presenting the same interface on both the CAB and host.

4 Protocol Implementation

We have implemented several transport protocols on the CAB, including TCP/IP and a set of Nectar-specific transport protocols. The Nectar-specific protocols provide datagram, reliable message, and request-response communication. The reliable message protocol is a simple stop-and-wait protocol, and the request-response protocol provides the transport mechanism for client-server RPC calls.

The implementation of the TCP/IP protocol suite serves as a good example of the use of the runtime system's features. Time-critical functions are performed by interrupt handlers and mailbox upcalls, most others by system threads. Mailboxes are used throughout for the management of data areas. The use of mailboxes proved advantageous in avoiding any copying of the data between receipt and presentation to the user. Although we only describe the implementation of TCP/IP, all the transport protocol implementations are structured in a similar fashion.

4.1 Internet Protocol

IP input processing is performed at interrupt time. When a packet arrives over the fiber, the datalink layer reads the datalink header and initiates DMA operations to place the data into an appropriate mailbox. For IP packets, this is always the IP input mailbox. After the entire protocol header arrives, the datalink layer issues a *start-of-data* upcall to the protocol so that useful work can be done while the remainder of the packet is being received into the mailbox. IP uses this opportunity to perform a sanity check of the IP header (including computation of the IP header checksum).

When the entire packet has been received, the datalink layer issues an *end-of-data* upcall. In this upcall, the IP input handler queues packets for reassembly if they are fragments of a larger datagram. The handler transfers complete datagrams to the input mailbox of the appropriate higher-level protocol. This transfer uses the mailbox `Enqueue` operation, so no data is copied.

Higher-level protocols (including ICMP) are required to provide an input mailbox to IP; this mailbox constitutes the entire receive interface between IP and higher protocols. One advantage to this interface is that it allows the higher protocols to be implemented either as mailbox upcalls, which are called whenever their input mailbox is written, or as separate threads, which block until the next packet arrives. In our current system, ICMP is implemented as a mailbox upcall, while UDP and TCP each have their own server threads.

While the receive interface between IP and higher protocols consists of a simple mailbox, the send interface is more complex. To send a packet, higher protocols are expected to call `IP_Output` with a header template, a reference to the data they wish to send, a flag indicating whether the data area should be freed once sent, and a route to the destination (if known). The header template must contain a partially filled-in IP header. Protocols may also append their own header to the end of the template. `IP_Output` fills in the remaining fields in the IP header and calls the datalink layer to transmit the packet.

4.2 Transmission Control Protocol

The Nectar TCP implementation runs almost entirely in system threads, rather than at interrupt time. This allows shared data structures to be protected with mutual exclusion locks rather than by disabling interrupts. We plan to compare this approach to a strictly interrupt-driven implementation of TCP as part of the experiment discussed in Section 3.1.

All TCP input processing is performed by the TCP input thread. This thread blocks on a `Begin_Get` until a packet arrives. Once it gets a packet, it examines the TCP header, checksums the entire packet, and performs standard TCP input processing. To pass data to the user, TCP simply deletes the headers and transfers the packet to the user's receive mailbox using the `Enqueue` operation.

A user wishing to send data on an established TCP connection places a request in the TCP *send-request* mailbox. The data to be sent may be placed in the send-request mailbox following the request, or it may already exist in some other mailbox, in which case the user includes a pointer to it in the request. The TCP send thread on the CAB services this request by placing the data on the send queue of the appropriate connection and calling the TCP output routine. CAB-resident senders can do this directly without involving the TCP send thread.

5 Usage

The flexibility of the CAB software architecture allows us to choose which layers in the protocol stack are handled by the CAB, effectively changing the interface the CAB presents to the host. Three such interfaces are described below, ordered in increasing degree of CAB functionality.

5.1 Network Device

The Nectar network can be used as a conventional, high-speed LAN by treating the CAB as a network device and enhancing the CAB device driver to act as a network interface. We have implemented a driver at this level for the Berkeley networking code [11], performing IP and higher-level protocols on the host as usual. The advantage of this approach is binary compatibility: all the familiar network services are immediately available.

To perform networking functions, the device driver cooperates with a server thread on the CAB that is responsible for transmitting and receiving packets over Nectar. The driver and the server share a pool of buffers: to send a packet the driver writes the packet into a free buffer in the output pool and notifies the server that the packet should be sent; when a packet is received the server finds a free input buffer, receives the packet into the buffer, and informs the driver of the packet's arrival.

5.2 Protocol Engine

The CAB can be used as a protocol engine by offloading transport protocol processing to the CAB. Several interfaces are possible on the host; these interfaces are independent of the particular transport protocols implemented on the CAB.

The Nectarine interface that was described in Section 3.5 provides applications with a flexible communication model. Since it uses the host-CAB buffering and synchronization facilities directly, some or all of CAB memory must be mapped into applications' address spaces.

The familiar Berkeley socket interface [13] is also being implemented at this level. Initially, an emulation library will be provided for applications that can be re-linked. Eventually, we will move this support into the UNIX kernel, which will intercept operations on Nectar connections and dispatch them to the CAB. This approach incurs the cost of system calls, but allows binary compatibility.

Work is also in progress to support the Mach interprocess communication interface [1]. Network IPC in Mach is provided by a message-forwarding server external to the Mach kernel; this server is a natural candidate for execution on the CAB.

5.3 Application-level Communication Engine

The Nectar CAB and its runtime system are more flexible than many proposed protocol engines since application-specific code can be executed on the CAB. Distributed applications on Nectar often perform tasks on both hosts and CABs, effectively using the CABs as application-specific network interfaces.

Common paradigms for parallel processing, such as divide-and-conquer and task-queue models, have been implemented on Nectar, using one or more CABs to divide the labor and gather the results. Examples include Noodles [6], a package for solid modeling; COSMOS [3], a switch-level circuit simulator; and Paradigm [5], a database for vision. Further work on load-balancing strategies and language-level tools for parallel program generation is in progress.

We are investigating several other areas that can make good use of the flexibility of the CAB:

- Communication is a major bottleneck in the Camelot distributed transaction system [16], so experiments are being planned to offload Camelot's distributed locking and commit protocols to the CAB.

Protocol	host to host	CAB to CAB
datagram	325	179
reliable message	414	241
request-response	438	287
UDP	842	536

Table 1: Roundtrip times in microseconds

- Using Mach together with Nectar, we are investigating network shared memory [9]. The CABs will run external pager tasks that cooperate to provide the required consistency guarantees.
- Research is under way to use the CAB to offload presentation layer functionality, such as the marshaling and unmarshaling of data required by remote procedure call systems [15].

6 Performance

A prototype Nectar system has been operational since January 1989. Currently the prototype system consists of 2 HUBs and 26 hosts in full-time use. This section reports the performance we have achieved on the current system. All measurements involving hosts were obtained using Sun 4 workstations.

6.1 Latency

Roundtrip latency times for UDP and for the Nectar-specific protocols are shown in Table 1.

Figure 6 shows a breakdown of the time taken to send a message between two host processes using the Nectar-specific datagram protocol. The fiber and Hub latency, less than 5 μsec, is not included. About 40% of the time is spent in the host-CAB interface at the sender and receiver, 40% is due to CAB-to-CAB latency, and the remaining 20% is spent on the host creating and reading the message. The time spent in the host-CAB interface is relatively large because each read or write over the VME bus takes about 1 μsec. More time is spent on the sending side, since the CAB must be interrupted and a CAB thread must be scheduled to handle the message. On the receiving side, the host process is polling for receipt of the message, so no interrupt or context switch is required.

6.2 CAB-to-CAB Throughput

The graph in Figure 7 shows throughput rates achieved between two CAB threads using the on-CAB implementation of TCP/IP and the Nectar reliable message protocol (RMP). For small packets (up to 256 bytes), the per-packet overhead dominates the transmission time completely, and the throughput doubles when the packet size doubles. For larger packet

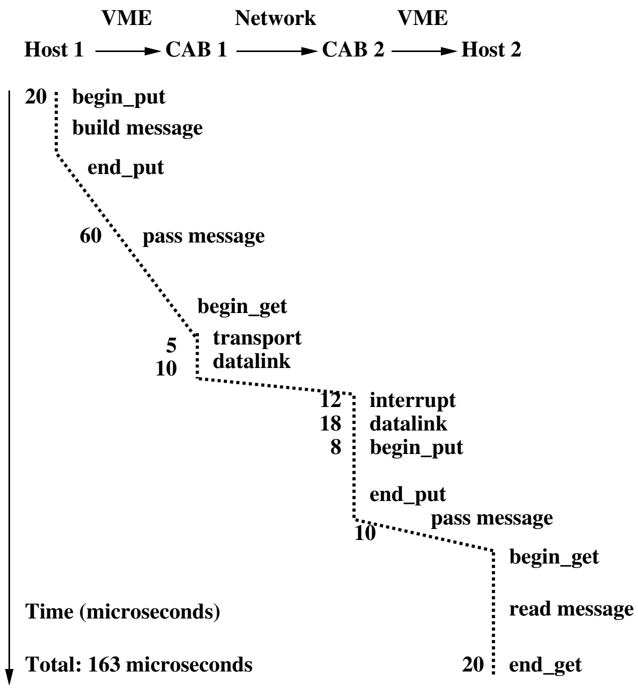

Figure 6: One-way host-to-host datagram latency analysis

sizes, the transmission time dominates and the throughput increase drops. The performance difference between TCP/IP and RMP is mostly due to the cost of doing TCP checksums in software. RMP does not do any software checksum computation but relies on the CRC implemented by the CAB hardware. The curve labeled “TCP w/o checksum” in Figure 7 shows that TCP without checksums is almost as fast as RMP.

6.3 Host-to-Host Throughput

The graph in Figure 8 shows throughput rates achieved between two host processes using the on-CAB implementation of TCP/IP and the Nectar reliable message protocol. The curves have the same shape as the CAB-CAB throughput curves, but they flatten earlier because the slow VME bus makes the transmission times more significant. The throughput of both protocols is limited by the bandwidth of the VME bus, which is about 30 Mbit/sec. The maximum TCP/IP bandwidth is about 24 Mbit/sec. In comparison, the maximum host-to-host throughput achieved when the CAB is acting as a simple network interface (described in Section 5.1) is 6.4 Mbit/sec. The same hosts can do better using Ethernet—achieving 7.2 Mbit/sec—because the on-board Ethernet interfaces bypass the VME bus.

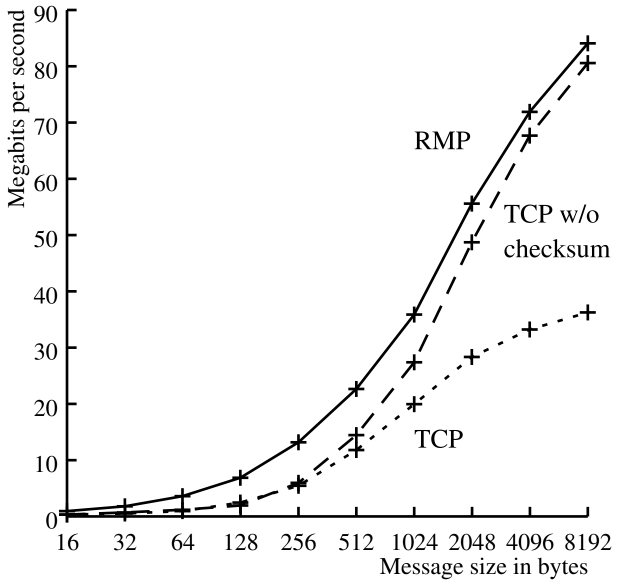

Figure 7: CAB-to-CAB throughput

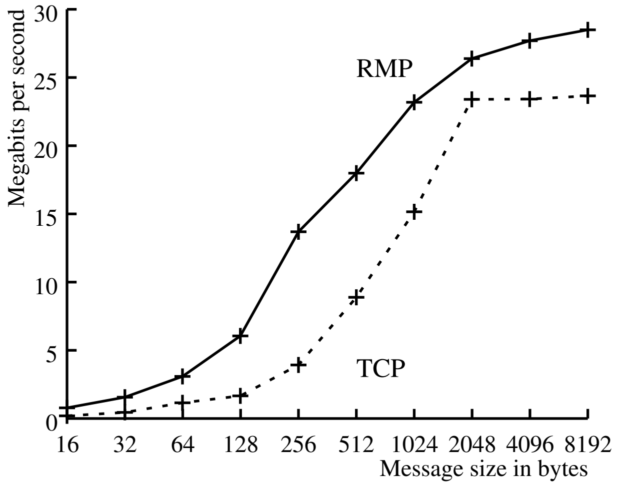

Figure 8: Host-to-host throughput

7 Conclusion

We have described the design and performance of the runtime system software for the Nectar communication processor. The flexibility of the runtime system allows application-specific communication tasks as well as transport protocols to be executed efficiently on the Nectar CAB.

The facilities provided by the runtime system allow protocols to be implemented easily and efficiently. The fact that we achieve roundtrip times as low as 325 μsec (for the datagram protocol) between two UNIX processes shows that the performance of the system has not been compromised by its flexibility.

The limiting factor in host-to-host performance over Nectar is the VME bus, the most common bus in our environment. Although we can obtain a throughput of 90 Mbit/sec between threads on two communication processors, we are unable to achieve more than 30 Mbit/sec between host processes. Although we optimized portions of the host-CAB interface to minimize the effect of the VME bottleneck, the overall design of the Nectar software is independent of the choice of bus. The software architecture described in this paper relies only on the presence of shared memory between the host and CAB, and we expect that it will perform well when higher-speed buses are used to connect the host and the CAB.

The Nectar system is now used every day by application developers. Our future work will include further performance evaluation and tuning, improving the structure and functionality of the CAB runtime system, and porting important applications such as NFS and the X Window System to Nectar.

8 Acknowledgments

The Nectar project is a team effort led by Professor H. T. Kung. The Nectar hardware was designed and built by Emmanuel Arnould, Matthieu Arnould, François Bitz, Onat Menzilcioğlu, and Steven Schlick. In addition to the authors, the following people have made significant contributions to the Nectar software: Michael Browne, Bernd Bruegge, Chiun-Hong Chien, Fred Christianson, Guy Jacobson, and I-Chen Wu.

References

[1] Michael J. Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young.
Mach: A new kernel foundation for UNIX development.
In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[2] Emmanuel A. Arnould, François J. Bitz, Eric C. Cooper, H. T. Kung, Robert D. Sansom, and Peter A. Steenkiste.
The design of Nectar: A network backplane for heterogeneous multicomputers.
In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 205–216, April 1989.
Also available as Technical Report CMU-CS-89-101, School of Computer Science, Carnegie Mellon University.

[3] Randal E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler.
COSMOS: A compiled simulator for MOS circuits.
In *24th Design Automation Conference*, pages 9–16. ACM and IEEE, 1987.

[4] Greg Chesson.
Protocol engine design.
In *Proceedings of the Summer 1987 USENIX Conference*, pages 209–215, June 1987.

[5] Chiun-Hong Chien and Long-Ji Lin.
Paradigm: An architecture for distributed vision processing.
In *Proceedings of the 10th International Conference on Pattern Recognition*. IEEE, June 1990.

[6] Young Choi.
Vertex-based Boundary Representation of Non-Manifold Geometric Models.
PhD thesis, Carnegie Mellon University, 1990.

[7] David D. Clark.
The structuring of systems using upcalls.
In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180. ACM, December 1985.

[8] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, June 1988.

[9] Allesandro Forin, Joseph Barrera, and Richard Sanzi.
The shared memory server.
In *Proceedings of the Winter 1989 USENIX Conference*, pages 229–243, January 1989.

[10] Hemant Kanakia and David R. Cheriton.
The VMP network adapter board (NAB): High-performance network communication for multiprocessors.
In *Proceedings of the SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 175–187. ACM, August 1988.

[11] 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, Reading, Massachusetts, 1989.

[12] Larry Peterson, Norman Hutchinson, Sean O'Malley, and Herman Rao.
The *x*-kernel: A platform for accessing Internet resources.
Computer, 23(5):23–33, May 1990.

[13] John S. Quarterman, Abraham Silberschatz, and James L. Peterson.
4.2BSD and 4.3BSD as examples of the UNIX system.
ACM Computing Surveys, 17(4):379–418, December 1985.

[14] J. H. Reppy.
Synchronous operations as first-class values.
In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 250–259, June 1988.

[15] Ellen H. Siegel and Eric C. Cooper.
Implementing OSI presentation layer functionality for parallel systems.
In *3rd Workshop on Large Grain Parallelism*. Software Engineering Institute, Carnegie Mellon University, October 1989.

[16] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Daniel J. Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson.
The Camelot project.
Database Engineering, 9(4), December 1986.
Also available as Technical Report CMU-CS-86-166, School of Computer Science, Carnegie Mellon University.