

Evolution of the Virtual Interface Architecture



The recent introduction of the VIA standard for cluster or system-area networks has opened the market for commercial user-level network interfaces. The authors examine how design decisions in prototype interfaces have helped shape this industry standard.

Thorsten
von Eicken

Werner
Vogels

Cornell
University

To provide a faster path between applications and the network, most researchers have advocated removing the operating system kernel and its centralized networking stack from the critical path and creating a *user-level network interface*. With these interfaces, designers can tailor the communication layers each process uses to the demands of that process. Consequently, applications can send and receive network packets without operating system intervention, which greatly decreases communication latency and increases network throughput.

Unfortunately, the diversity of approaches and lack of consensus has stalled progress in refining research results into products—a prerequisite to the widespread adoption of these interfaces. Recently, however, Intel, Microsoft, and Compaq have introduced the Virtual Interface Architecture,¹ an emerging standard for cluster or system-area networks. Products based on the VIA have already surfaced, notably GigaNet's GNN1000 network interface (<http://www.giganet.com>). As more products appear, research into application-level issues can proceed and the technology of user-level network interfaces should mature.

Several prototypes—among them Cornell University's U-Net²—have heavily influenced the VIA. In this article, we describe the architectural issues and design trade-offs at the core of these prototype designs, including

- How to provide applications with direct access to the network interface hardware, yet retain sufficient protection to ensure that applications can't interfere with each other.
- How to design an efficient, yet versatile programming interface. Applications must be able to access the network interface and still control buffering, scheduling, and addressing. The programming interface, on the other hand, must

accommodate a wide variety of hardware implementations.

- How to manage resources, in particular memory. Applications must consider the costs of DMA transfers that map a virtual address to a physical one. At the same time, implementation-specific details must be hidden from the application, and the operating system must ultimately control all resource allocation.
- How to manage fair access to the network without a kernel path which, in traditional protocol stacks, acts as a central point of control and scheduling.

PERFORMANCE FACTORS

Network performance is traditionally described by the bandwidth achieved when an infinite stream is transmitted. However, an increasing number of applications are more sensitive to the network's round-trip time, or *communication latency*, and the bandwidth when sending many small messages.

Low communication latency

Communication latency is due mainly to processing overhead—the time the processor spends handling messages at the sending and receiving ends. This may include managing the buffers, copying the message, computing checksums, handling flow control and interrupts, and controlling network interfaces.

As Figure 1 shows, the round-trip times for a remote procedure call with a user-level network interface can actually be lower than for a local RPC under Windows NT. The remote call uses Microsoft's distributed component object model (DCOM) with the VIA on the GNN1000 interface. The local call uses Microsoft's component object model (COM). The figure also shows the round-trip latency of raw messages over the VIA, which we used as a baseline in estimating the DCOM protocol overhead.

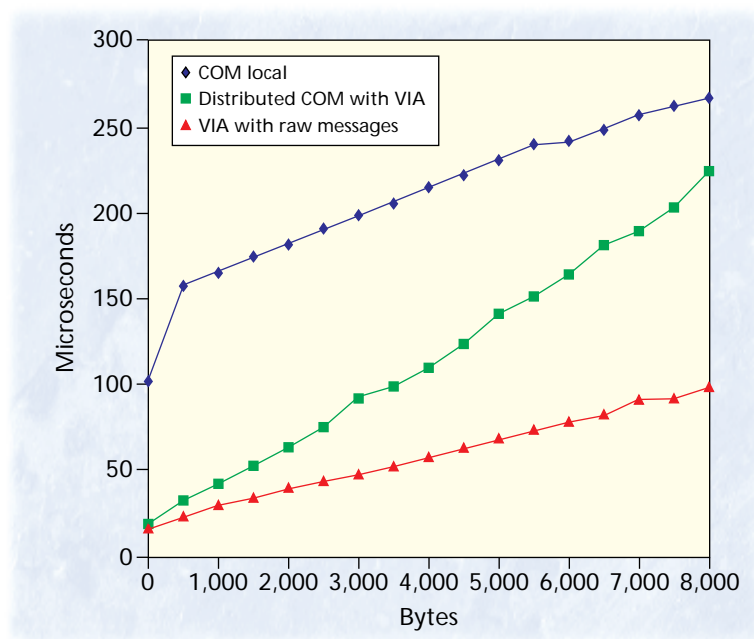


Figure 1. Round-trip times for a local remote procedure call (LRPC) under Windows NT (COM local) versus a remote procedure call over the Virtual Interface Architecture using GigaNet's GNN1000 interface (Distributed COM with VIA). By bypassing the kernel the remote call is actually faster than the local one. The round-trip latency of raw messages over the VIA (VIA with raw messages) shows that there is room for improvement in the DCOM protocol.

Low communication latency is key to using clusters in enterprise computing, where systems must be highly available and scalable. Cluster management and cluster-aware server applications rely on multiround protocols to reach agreement on the system's state when there are potential node and process failures. These protocols involve multiple participants—all of which must respond in each round. This makes the protocols extremely sensitive to any latency. Cluster applications that require fault tolerance (for example through a primary/backup scheme or through active replication) use extensive intracenter communication to synchronize replicated information internally. These cluster applications can be scaled up only if the intracenter communication is many times faster than the time in which the systems are expected to respond to their clients. Recent experiments with Microsoft's Cluster Server have shown that without low-latency intracenter communication, the scalability is limited to eight nodes.³

On the parallel computing front, many researchers use networks of workstations to provide the resources for computationally intensive parallel applications. However, these networks are difficult to program, and the communication costs across LANs must decrease by more than an order of magnitude to address this problem.

High bandwidth for small messages

The demand for high bandwidth when sending many small messages (typically less than 1 Kbyte each) is increasing for the same reasons industry needs low communication latency. Web servers, for example, often receive and send many small messages to many clients. By reducing the per-message overhead, user-level network interfaces attempt to provide full network bandwidth for the smallest messages possible.

Reducing the message size at which full bandwidth can be achieved may also benefit datastream proto-

cols like TCP, whose buffer requirements are directly proportional to the communication latency. The TCP window size, for example, is the product of the network bandwidth and the round-trip time. One way to have maximum bandwidth at minimum cost is to achieve low latency in local area networks, which will keep buffer consumption within reason.

Flexible communication protocols

The traditionally strict boundary between applications and protocols has made it harder to develop more efficient networked applications. Two issues are central: integrating the application and protocol buffer management, and optimizing the protocol control path.

In traditional systems, the lack of support for integrated buffer management accounts for a significant part of the processing overhead. This affects application-specific reliability protocols—RPCs, in particular—which must keep data for retransmission. Without shared buffer management, for example, designers cannot use reference count mechanisms to lower the overhead from copying and from transferring buffers between the application and the kernel.

There are several ways to build flexible, optimal communication protocols. Designers can use experimental or highly optimized versions of traditional protocols when integrating the code path of generic and application-specific protocols. Advanced protocol design techniques include *application-level framing*, in which the protocol buffering is fully integrated with application-specific processing, and *integrated-layer processing*, in which many protocol layers are collapsed into highly efficient, monolithic code paths. Designers can also use a compiler to compile all protocols together instead of just a small subset of application-specific protocols. Finally, new techniques, such as fast-path generation in the Ensemble communication system, use formal verification technology to automatically generate optimized protocol stacks. These techniques rely on control over the complete protocol stack, including the lowest layers, and require the stack to run in a single address space.

MILESTONES IN INTERFACE DESIGN

Message-based user-level network interfaces let applications exchange data by sending and receiving explicit messages, similar to traditional multicomputer message-passing interfaces, such as MPI, Intel's NX, and Thinking Machines' CMMD. All the user-level network interfaces we describe let multiple users on a host access the network simultaneously. To provide protection, they separate the communication setup from data transfer. During setup, the operating system is involved and performs protection checks to ensure that applications cannot interfere with each other. During data transfer, the interface bypasses the

operating system and performs simple checks to enforce protection.

Figure 2 shows system memory with two applications accessing the network through a user-level network interface. A device driver in the operating system controls the interface hardware in a traditional manner and manages the application's access to it. Applications allocate message buffers in their address space and call on the device driver to set up their access to the network interface. Once set up, they can initiate transmission and reception and the interface can transfer data to and from the application buffers directly using direct memory access.

User-level network interface designs vary in the interface between the application and the network—how the application specifies the location of messages to be sent, where free buffers for reception get allocated, and how the interface notifies the application that a message has arrived. Some network interfaces, such as Active Messages or Fast Messages, provide send and receive operations as function calls into a user-level library loaded into each process. Others, such as U-Net and VIA, expose per-process queues that the application manipulates directly and that the interface hardware services.

Parallel computing roots

Message-based user-level network interfaces have their roots in traditional multicomputer message-passing models. In these models, the sender specifies the data's source memory address and the destination processor node, and the receiver explicitly transfers an incoming message to a destination memory region. Because of the semantics of these send and receive operations, the user-level network interface library must either buffer the messages (messages get transferred via the library's intermediate buffer) or perform an expensive round-trip handshake between the sender and receiver on every message. In both cases, the overhead is high. *Active Messages*⁴ were created to address this overhead. Designs based on this notion use a simple communication primitive to efficiently implement a variety of higher level communication operations. The main idea is to place the address of a dedicated handler into each message and have the network interface invoke the handler as soon as the interface receives that message. Naming the handler in the message promotes very fast dispatch; running custom code lets the data in each message be integrated into the computation efficiently.

Thus, Active Message implementations did not have to provide message buffering and could rely on handlers to continually pull messages out of the network. Active Message implementations also did not need to provide flow control or retransmit messages because the networks in the parallel machines already implemented these mechanisms.

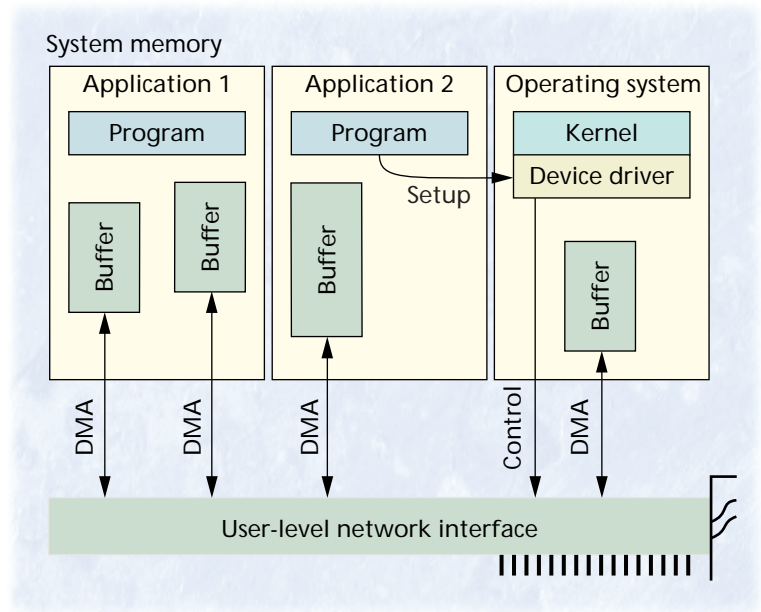


Figure 2. System memory with two applications accessing the network using a user-level network interface. A conventional device driver in the operating system controls the interface hardware. The device driver also sets up direct access to the interface for applications (setup). The applications can then allocate buffers in their address space and have the interface transfer message data into and out of these buffers directly using direct memory access (DMA).

Although Active Messages performed well on the first generation of commercial massively parallel machines, the *immediate* dispatch on arrival became more and more difficult to implement efficiently on processors with deepening pipelines. Some implementations experimented with running handlers in interrupts, but the increasing interrupt costs caused most of them to rely on implicit polling at the end of sends and on explicitly inserted polls. Thus, the overhead problem resurfaced, since polling introduces latency before message arrival is detected and incurs overhead even when no messages arrive.

Illinois *Fast Messages*⁵ addressed the immediacy problem by replacing the handler dispatch with buffering and an explicit poll operation. With buffering, Fast Messages can delay running the handlers without backing up the network, and applications can reduce the frequency of polling, which in turn reduces overhead. The send operation specifies the destination node, handler, and data location. The Fast Message transmits the message and buffers it at the receiving end. When the recipient calls are extracted, the Fast Message implementation runs the handlers of all pending messages.

By moving the buffering back into the message layer, the layer can optimize buffering according to the target machine and incorporate the flow control needed to avoid deadlocks and provide reliable communication over unreliable networks. Letting the message layer buffer small messages proved to be very effective—it was also incorporated into the U.C. Berkeley Active Messages II (AM-II) implementations—as long as messages could be transferred unbuffered to their final destination.

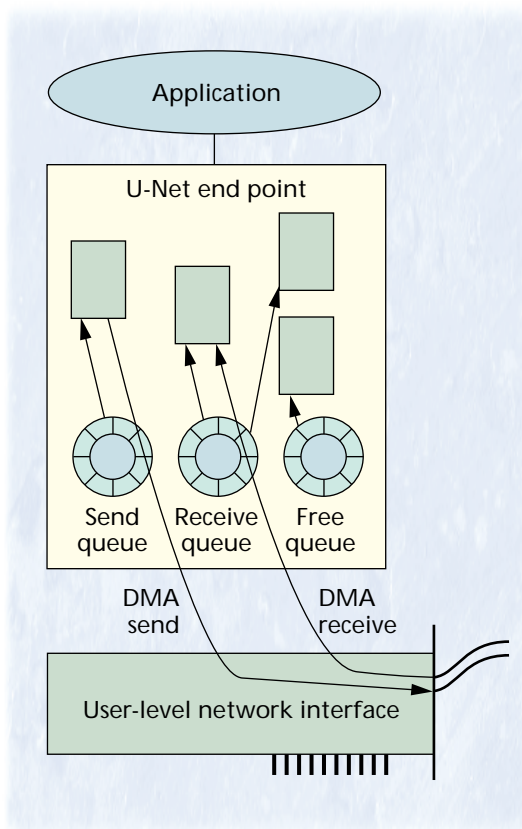
U-Net

U-Net was the first design to significantly depart from both Active Messages and Fast Messages. Table 1 summarizes the performance of U-Net and four subsequent designs as implemented on Myricom's

Table 1. Round-trip performance of standard sockets and user-level network interfaces on Myricom's Myrinet network.

	Myrinet	U-Net	VMMC-2	AM-II	Fast Messages	VIA
Latency (μ sec)	250	13	11	21	11	60
Bandwidth (Mbytes/s)	15	95	97	31	78	90

Figure 3. U-Net architecture. Each application accesses the network through U-Net end points. An end point contains send, receive, and free queues that point to buffers the application has allocated. The interface hardware accesses the queues to determine which buffer to transfer messages into, for reception (DMA receive), and out of, for transmission (DMA send).



Myrinet network.⁶ U-Net provides an interface to the network that is closer to the functionality typically found in LAN interface hardware. It does not allocate any buffers, perform any implicit message buffering, or dispatch any messages. Instead of providing a set of API calls, as in previous interfaces, it consists of a set of in-memory queues that convey messages to and from the network. In essence, Active Messages and Fast Messages define a very thin layer of software that presents a uniform interface to the network hardware. U-Net, on the other hand, specifies the hardware's operation so that the hardware presents a standard interface directly to user-level software.

The U-Net approach offered the hope of a better fit into a cluster environment. Such an environment typically uses standard network technology such as Fast Ethernet or Asynchronous Transfer Mode, and the network must handle not only parallel programs, but also more traditional stream-based communication. Machines within the cluster communicate with outside hosts using the same network.

Architecture. In U-Net, end points serve as an application's handle into the network and contain three circular message queues, as Figure 3 shows. The queues hold descriptors for message buffers that are to be sent (send queue), are free to be received (free queue), and have been received (receive queue). To send a message, a process queues a descriptor into the send queue. The descriptor contains pointers to the message buffers, their lengths, and a destination address. The network interface picks up the descriptor, validates the destination address, translates the virtual buffer addresses to physical addresses, and transmits the data using direct memory access (DMA).

When the network interface receives a message, it determines the correct destination end point from the header, removes the needed descriptors from the associated free queue, translates their virtual addresses, transfers the data into the memory using DMA, and queues a descriptor into the end point's receive queue. Applications can detect the arrival of messages by polling the receive queue, by blocking until a message arrives (as in a Unix select system call), or by receiving an asynchronous notification (such as a signal) when the message arrives.

Protection. In U-Net, processes that access the network on a host are protected from one another. U-Net maps the queues of each end point only into the address space of the process that owns the end point, and all addresses are virtual. U-Net also prevents processes from sending messages with arbitrary destination addresses and from receiving messages destined to others. For this reason, processes must set up communication channels before sending or receiving messages. Each channel is associated with an end point and specifies an address template for both outgoing and incoming messages. When a process creates a channel, the operating system validates the templates to enforce system-specific policies on outgoing messages and to ensure that all incoming messages can be assigned unambiguously to a receiving end point.

The exact form of the address template depends on the network substrate. In versions for ATM, the template simply specifies a virtual channel identifier; in versions for Ethernet, a template specifies a more elaborate packet filter.

U-Net does not provide any reliability guarantees beyond that of the underlying network. Thus, in general, messages can be lost or can arrive more than once.

AM-II and VMMC

Two models provide communication primitives inspired by shared memory: the AM-II, a version of Active Messages developed at the University of California at Berkeley,⁷ and the Virtual Memory Mapped Communication model, developed at Princeton University as part of the Shrimp cluster project.⁸ The primitives eliminate the copy that both fast messages and U-Net typically require at the receiving end. Both Fast Message implementations and U-Net receive messages into a buffer and make the buffer available to the application. The application must then copy the data to a final destination if it must persist after the buffer is returned to the free queue.

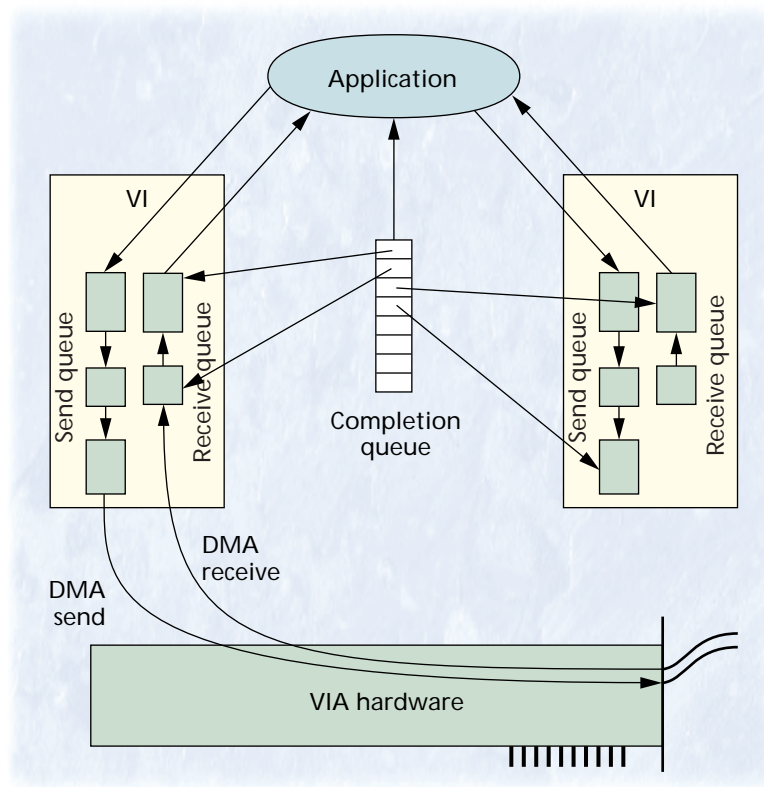
AM-II provides `put` and `get` primitives that let the initiator specify the addresses of the data at the remote end: `put` transfers a local memory block to a remote address; `get` fetches a remote block. VMMC provides a primitive essentially identical to `put`. In all these primitives, no receiver intervention is necessary to move the data to the final location as long as the sender and receiver have previously coordinated the remote memory addresses. AM-II associates an Active Message handler with each `put` and `get`; VMMC provides a separate notification operation.

VMMC requires that communicating processes pin down all memory used for communication so that it cannot be paged. VMMC-2⁹ lifts this restriction by exposing memory management as a *user-managed translation look-aside buffer*. Before using a memory region for sending or receiving data the application must register the memory with VMMC-2, which enters translations into the UTLB. VMMC-2 also provides a default buffer into which senders can transmit data without first asking the receiver for a buffer address.

Virtual Interface Architecture

The VIA combines the basic operation of U-Net, adds the remote memory transfers of VMMC, and uses VMMC-2's UTLB. Processes open *virtual interfaces* (VI) that represent handles onto the network, much like U-Net's end points. As Figure 4 shows, each VI has two associated queues—send and receive—that are implemented as linked lists of message descriptors. Each descriptor points to one or multiple buffer descriptors. To send a message an application adds a new message descriptor to the end of the send queue. After transmitting the message, the VIA sets a completion bit in the descriptor, and the application eventually takes the descriptor out of the queue when it reaches the queue's head. For reception, the application adds descriptors for free buffers to the end of the receive queue, which VIA fills as messages arrive.

Each VI represents a connection to a single other remote VI. This differs from the U-Net end point, which can aggregate many channels. In the VIA, a process can



create one or more *completion queues* and associate each with multiple VIs. The network interface fills entries in the completion queue to point to entries in the send or receive queue that have been fully processed.

The VIA also provides direct transfers between local and remote memory. These *remote DMA* writes and reads are similar to AM-II's `puts` and `gets` and VMMC's transfer primitives.

To provide some protection, the VIA lets a process specify which regions of its memory are available for RDMA operations. Memory management in the VIA is very similar to the VMMC-2. A UTLB resides in the network interface. All memory used for communication must be registered with the VIA before it is used, including all queues, descriptors, and buffers. Registering a region returns a handle, and all addresses must be specified using the appropriate region handle and a virtual address.

DESIGN TRADE-OFFS

As the previous description shows, the user-level network interface designs that have shaped the VIA differ from each other in many respects. These differences reflect attempts to optimize against network hardware and programming environments. Trade-offs are most apparent in the choice of queue structure, memory management strategy, and message multiplexing approach.

Figure 4. Queuing in the Virtual Interface Architecture. Each application can open multiple virtual interfaces (VIs), each with its own send and receive queues. Each VI is associated with a completion queue to which the VIA adds a descriptor for every completed transmission or reception. By sharing one completion queue across many VIs, an application must check only a single queue to pick up all events related to its network connections.

Integrating buffer management between the application and the network interface is important in eliminating data copies and reducing allocations and deallocations.

Queue structure

The queues in U-Net and the VIA expose a structure very similar to most hardware network devices. Exposing the queues instead of providing a procedural interface (as in Fast Messages and Active Messages) means that higher software layers can manage the required descriptors and buffers directly. This, in turn, means that at no point is an application's executing thread required to yield control to complete an operation.

Exposing the queues also very naturally splits an operation's initiation from its completion. Pushing a descriptor into the send queue initiates the transmission; the network interface signals completion using the descriptor. The actual transmission or reception can thus occur asynchronously to the ongoing computation.

The queues are designed to accept *scatter-gather* descriptors, which allow applications to construct messages from a collection of noncontiguous buffers. For example, in a sliding window protocol (in which the transmission window moves over a large data buffer), the protocol implementation places its information in a separate buffer. The network interface then concatenates the data and protocol information.

Memory management

Integrating buffer management between the application and the network interface is important in eliminating data copies and reducing allocations and deallocations. This integration gives rise to additional complexity, however, because the application uses virtual addresses for its message buffers, whereas the network interface DMA engine requires physical memory addresses for its transfers. Thus, a trusted entity must translate a virtual address to a physical one each time a buffer is handed to the network interface. In addition, the operating system must track the memory pages available to the interface for DMA transfers so that the operating system can keep mappings constant.

The VIA's memory management is inspired by VMMC-2's UTLB, which makes the application responsible for managing virtual-to-physical address translations. The application must request a mapping for a memory region from the operating system before using any buffers in that region. The advantage is that the operating system can install all mappings in the network interface, which can then translate all virtual addresses using a simple lookup. The primary drawback is that the application must take care to manage its memory regions judiciously. In simple cases, a fixed set of buffers is allocated, so it is easy to keep the memory region handles in the buffer descriptors. However, if the application transmits data out of internal data structures directly, it may have to request a new translation for every transmission, which is time-consuming.

In U-Net, in contrast, applications can place buffers anywhere in their virtual address space and the network interface does the virtual-to-physical address translation.¹⁰ For this reason, the interface incorporates a TLB that maps `<process ID, virtual address>` pairs to physical page frames and read/write access rights.

The disadvantage of U-Net's approach is that the application may present an address that has no mapping in the interface's TLB. If the address is for a buffer to be sent, the interface must request a translation from the host kernel, which requires a somewhat complex and costly handshake (approximately 20 μ s of overhead in Windows NT, for example).

To avoid TLB misses when a message arrives, the interface must pretranslate entries in each free buffer queue. If a message arrives and no pretranslated buffer is available, the interface drops the message. Requesting translation on-the-fly would be very difficult because reception occurs within an interrupt handler, which has only limited access to kernel data structures.

An important benefit of U-Net's memory-allocation scheme is that the host kernel can limit the amount of memory pinned down for the network by limiting the number of valid entries in the interface's TLB. This is somewhat analogous to preallocating and pinning a fixed-size buffer pool in kernel-based network stacks. The difference is that, in U-Net, the set of pages in the buffer pool can vary over time.

Shifting control over memory management from the operating system to the application, as the VIA does, has a potentially more serious drawback. If an application requests a mapping, the operating system must grant it or risk an application failure. Once it grants a mapping, the operating system cannot revoke it to shift resources, for example, to a higher priority process. If servers and workstations host multiple applications with simultaneous open network connections, and each application uses, say 200 to 300 Kbytes of buffer space, a significant fraction of the physical memory must be dedicated to network activity.

In U-Net, in contrast, only entries in the interface's TLB must be pinned in memory. The interface can also remove mappings of applications not actively using their network buffers, thereby making the memory eligible for paging and available to applications in general.

Multiplexing and demultiplexing

In user-level network interfaces, the network interface must demultiplex arriving messages onto the correct receive queue. Similarly, when sending a message, the interface must enforce protection (by validating addressing information, for example). The complex-

ity of both operations depends heavily on the type of network used.

The VIA's approach to multiplexing is strictly connection oriented. Two virtual interfaces must set up a connection before any data can be transmitted. The specification does not prescribe how the connection must be represented at the network level, so designers could run the VIA over an Ethernet or even over the Internet Protocol by defining some handshake between the end points and by using a special field to differentiate connections.

Because of the VIA's connection orientation, applications that use VIA networks cannot interoperate with applications that use non-VIA networks. For example, a server cluster using the VIA internally must use conventional network interfaces to communicate with clients outside the cluster.

U-Net, in contrast, makes the multiplexing and demultiplexing processes much more flexible. In the Fast Ethernet version, the network interface implements a full packet filter, which is not connection oriented. When an application opens a communication channel, it specifies patterns for parsing the headers of incoming and outgoing packets. The kernel agent validates the patterns to ensure that there will be no conflict with other applications and enters them into the multiplexing data structures. The generic nature of U-Net's packet processing means that communication with a generic protocol stack is possible, as is the sending and receiving of multicast messages.

The filtering must be implemented as a dynamic packet parser to support a variety of network protocols and accommodate the nature of identifying fields, which are not at fixed locations in messages. The packet filter must be more sophisticated than those used previously¹¹ because there is no backup processing path through the operating system kernel for handling unknown packets. Such a fail-safe path is not an option because the architecture would then force the operating system to jointly manage the receive queues with the interface. The Fast Ethernet version of U-Net sidesteps the issue by using a standard network interface and implementing its functionality in the interrupt handler, where the complexity of the packet filter is tolerable.

Remote memory access

Handling RDMA reads and writes adds considerable complexity to VIA implementations. When receiving a write, the interface must not only determine the correct destination VI but also extract the destination memory address from the message and translate it. The main issue here is how to handle transmission errors correctly: The interface must verify the packet checksum before any data can be stored into memory to ensure that no error affected the des-

tinuation address. This verification is not trivial because most networks place the checksum at the end of the packet, and the interface must buffer the entire packet before it can start the DMA transfer to main memory. Alternatively, a separate checksum could be included in the header itself.

RDMA reads also present a problem. The network interface may receive read requests at a higher rate than it can service, and it will have to queue the requests for later processing. If the queue fills up, requests must eventually be dropped. To ensure a safe implementation of RDMA reads, the VIA restricts read requests to implementations that provide reliable delivery.

VIA designers succeeded in picking most of the cherries from extensive research in user-level interfaces and in providing a coherent specification. Users familiar with this research will have little difficulty in adapting to the VIA and will enjoy the benefit of multiple off-the-shelf commercial hardware implementations. Because the VIA's primary target is server area networks, however, its designers did not provide one important cherry from U-Net: The VIA does not provide interoperability with hosts that use conventional network interfaces.

To fully exploit the promise of the VIA and of fast networks in general, significant additional research is needed in how to design low-latency protocols, marshal objects with low overhead, create efficient protocol timers, schedule communication judiciously, and manage network memory. Memory management presents particularly difficult trade-offs. The UTLBs in VMMC-2 and the VIA are easy to implement but take away resources from the general paging pool. The TLB proposed in U-Net implements a close interaction with the operating system and provides fair memory management but is more complex and incurs extra costs in the critical path whenever a miss occurs.

Thus, although the VIA user-level network interfaces are now available to everyone, many design issues must yet be solved before application writers can enjoy painless benefits. Hopefully, research will break through some of these issues as users begin to document their experiences in using VIA. ❖

The VIA's approach to multiplexing is strictly connection-oriented. Two virtual interfaces must set up a connection before any data can be transmitted.

Acknowledgments

This research is funded by the US Defense Advanced Research Projects Agency's Information Technology Office contract ONR-N00014-92-J-1866, NSF contract CDA-9024600, a Sloan Foundation fellowship, and Intel Corp. hardware donations.

References

1. D. Dunning et al., "The Virtual Interface Architecture," *IEEE Micro*, Mar.-Apr. 1998, pp. 66-76.
2. T. von Eicken et al., "U-Net: A User-level Network Interface for Parallel and Distributed Computing," *Proc. 15th Symp. Operating System Principles*, ACM Press, New York, 1995, pp. 40-53.
3. W. Vogels et al., "Scalability of the Microsoft Cluster Service," *Proc. Second Usenix Windows NT Symp.*, Usenix Assoc., Berkeley, Calif., 1998, pp. 11-19.
4. T. von Eicken et al., "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 256-266.
5. S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proc. Supercomputing '95*, ACM Press, New York, 1995.
6. N.J. Boden et al., "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, Feb. 1995, pp. 29-36.
7. B.N. Chun, A.M. Mainwaring, and D.E. Culler, "Virtual Network Transport Protocols for Myrinet," *IEEE Micro*, Jan.-Feb. 1998, pp. 53-63.
8. M. Blumrich et al., "Virtual-Memory-Mapped Network Interfaces," *IEEE Micro*, Feb. 1995, pp. 21-28.
9. C. Dubnicki et al., "Shrimp Project Update: Myrinet Communication," *IEEE Micro*, Jan.-Feb. 1998, pp. 50-52.
10. M. Welsh et al., "Memory Management for User-Level Network Interfaces," *IEEE Micro*, Mar.-Apr. 1998, pp. 77-82.
11. M. Bailey et al., "Pathfinder: A Pattern Based Packet Classifier," *Proc. Symp. Operating Systems Design and Implementation*, Usenix Assoc., Berkeley, Calif., 1994, pp. 115-123.

Thorsten von Eicken is an assistant professor of computer science at Cornell University. His research interests are in high-performance networking and Internet server security. He received a PhD in computer science from the University of California at Berkeley. He is a member of the IEEE Computer Society and the ACM.

Werner Vogels is a research associate in computer science at Cornell University. His research targets high availability in distributed systems, with a particular focus on enterprise cluster systems. He is a member of the IEEE Computer Society and the ACM.

Contact Von Eicken at Dept. of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853; tve@cs.cornell.edu.