VIRTUAL INTERFACE ARCHITECTURE DRAFT WRITE−UP

Roberto Innocente and Olumide Sunday Adewale

INTRODUCTION

The recent arrivals of fast networks, such as Myrinet, asynchronous transfer mode (ATM) coupled with powerful personal computers and workstations, has allowed for cost effective high−performance and high−availability clusters to be built from commodity parts. Additionally, the new cluster platform offers enough raw hardware performance to allow execution of applications such as parallel scientific computing or parallel servers, which used to be run on massively parallelised processors (MPPs) and symmetric multi−processors (SMPs). However, to achieve good results, the cluster communication subsystem must deliver user−to−user performance at least comparable to that of MPPs.

These new networks have exposed many problems with the traditional implementation of communications software; dramatic increases in hardware speed have not been translated to high application performance. For example, the Myrinet network is capable of delivering about two orders of magnitude higher bandwidth than traditional Ethernet networks. The Myrinet implementation of the Berkeley socket interface, however, delivers only one order of magnitude higher bandwidth than the Ethernet implementations: a factor of 10 less than expected, due mainly to data copying and buffer management. Similarly, it has been demonstrated that end−to−end latency on new networks, such as ATM, can sometimes exceed that of old Ethernet platforms, because of processing overheads introduced by the new interface.

The performance problems of implementing old communication interfaces on top of the new networks, coupled with the need for a high−performance communication layer for new classes of applications, has motivated the emergence of user−space communication system. Previous work has shown that most overhead in traditional communication systems is caused by software, e.g., system calls, buffer management, and data copying. In order to address these problems, several thin user−space communication subsystem layers have been proposed by the research community and indutry, which eliminate system calls and reduce or eliminate buffer management overheads. These systems can be used to implement traditional compatibility libraries, compilers, and even applications code written directly by users. These systems include active messages (AM), virtual memory−mapped communication (VMMC), fast messages (FM,) U−Net, low−level application programming interface (LAPI), basic interface for parallelism (BIP), Trapeze, DBM, PM, MyriAPI and the like. All of these communication systems use much simpler communication protocols in comparison with legacy protocols such as the TCP/IP. The role of the operating system has been much reduced in these systems and in most cases user applications are given direct access to the network interface. However, usability and applicability are strongly affected by subtle issues in the design and implementation of these software layers. High performance often conflicts with other important goals, such as multiprogramming, protected communication, portability, message ordering, fault tolerance, availability and security.

Building on the ideas in academic research on user−level communication, Compaq, Intel, Microsoft have jointly developed the virtual interface architecture specification (). The virtual interface specification describes a network architecture that provides user−level data transfer. Since the introduction of virtual interface architecture, few software and hardware implementations of the virtual interface architecture have become available. The Berkeley VIA, Giganet VIA, Servernet VIA, M−VIA, Myrinet and FirmVIA () are among these implementations. Hardware implementations of VIA provide support VI context management, queue management, and memory mapping functions in addition to standard network communication functions while software implementations provide these functionalities in NIC firmware (Myrinet), in special purpose device drivers, or in an intermediate driver layered on top of the standard network driver.

MILESTONES IN INTERFACE DESIGN

Message−based user−level network interfaces let applications exchange data by sending and receiving explicit messages, similar to traditional multi−computer message−passing interfaces, such as MPI,. 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.

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 Origins

Message−based user−level network interfaces have their roots in traditional multi−computer 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* 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.

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 explicitly inserted polls. Thus, the overhead problem resurfaced, since polling introduces latency before 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. 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.

In U−Net, end points serve as an application's handle into the network and contain three circular message queues. 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.

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 and the Virtual Memory Mapped Communication model, developed at Princeton University as part of the Shrimp cluster project. 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−2 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.

Basic Interface for Parallelism (BIP)
BIP is a low−level communication layer for Myrinet network, which allows an efficient access to the hardware, with zero memory copy communications. It only implements a very link flow control policy. It is not meant to be directly used by the parallel programmer, since it provides a very low degree of functionality. BIP messages are implemented for cluster of workstations, linked by Myrinet boards with LANAI processor. The implement consists of a user−level library associated with a custom MCP that will run on the Myrinet board. MPI−BIP is a port of MPICH over Myrinet network using the BIP communication layer.

An extension of BIP, called BIP−video enables access directly to the video board from the network board without any processing from the host processor.

Low−Level Application Programming Interface (LAPI)
The LAPI is a non−standard application−programming interface designed to provide optimal communication performance on the SP Switch. It is based on an *active message* programming mechanism that provides a one−sided communications model (that is, one process initiates an operation and the completion of that operation does not require any other process to take a complementary action). The LAPI library provides the functions **PUT**, **GET**, and a general *active message* function that allows programmers to supply extensions by means of additions to the notification handlers. The LAPI is designed for use by libraries and power programmers for whom performance is more important than code portability.

LAPI is an asynchronous communication mechanism intended to provide users the flexibility to write parallel programs with dynamic and unpredictable communication patterns. LAPI is architected to be an efficient (low latency, high bandwidth) interface. In order to keep the LAPI interface as simple as possible it is designed with a small set of primitives. However the limited set does not compromise on

functionality expected from a communication API. LAPI functionality includes data communication as well as synchronization and ordering primitives. Further, by providing the active message function as part of the interface the LAPI design allows users to expand the functionality to suit their application needs.

Active Message (AM) was selected as the underlying infrastructure for LAPI. We use the term *origin* to denote the task (or process or processor) that initiates a LAPI operation, and the term *target* to denote the other task whose address space is accessed by the LAPI operation. The active message includes the address of a user−specified handler. When the active message arrives at the target process, the specified handler is invoked and executes in the address space of that process. Optionally, the active message may also bring with it a user header and data from the originating process. The user header contains parameters for use by the header handler in the target process. The data is the actual message the user intends to transmit from the origin to the target. The operation is unilateral in the sense that the target process does not have to take explicit action for the active message to complete. Buffering (beyond what is required for network transport) is not required because storage for arriving data (if any) is specified in the active message, or is provided by the invoked handler. The ability for users to write their own handlers provides a generalized yet efficient mechanism for customizing the interface to one's specific requirements. LAPI supports messages that can be larger than the size supported by the underlying
network layer. This implies that data sent using an active message call will arrive in multiple packets; further these packets can arrive out of order. This places some requirements on how the handler is written. When the active message brings with it data from the originating process, LAPI requires that the "handler" be written as two separate routines:

- A *header_handler* function: This is the function specified in the active message call. It is called when the first packet of the message arrives at the target, and it provides the LAPI dispatcher (a part of the LAPI layer that deals with the arrival of messages and invocation of handlers) with: a) an address where the arriving data of the message must be copied, and b) the address of the optional *completion* handler; and
- A *completion_handler* which will be called after the whole message has been received (i.e. all the packets of the message have reached the target process).

The decoupling of the handler into a header handler/completion handler in the active message infrastructure allows multiple independent streams of messages to be sent and received simultaneously within a LAPI context. At any given instance LAPI ensures that only one header handler per LAPI context is allowed to execute. The rationale for this decision is that the header handler is just expected to return a buffer pointer for the incoming message and locking overheads might be comparatively expensive. Further, while the header handler executes, no progress can be made on the network interface. Multiple completion handlers are allowed to execute concurrently per LAPI context (the user is responsible for any synchronization among the completion handlers).

Fig. 1 illustrates the flow of data and control in a LAPI active message. A process on the origin makes the LAPI_Amsend call. The call initiates a transfer of the header *uhdr* and data *udata* at the origin process to the target process specified in the LAPI active message call. As soon as the user is allowed to reuse *uhdr* and *udata*, an indication is provided via *org_cntr* at the origin process. At some point (Step 1) the header and data arrive at the target. On arrival at the target, an interrupt is generated which results in the invocation of the LAPI dispatcher. The LAPI dispatcher identifies the incoming message as a new message and calls the *hdr_hndlr* specified by the user (Step 2) in the LAPI active message call. The handler returns a buffer pointer where the incoming data is to be copied (Step 3). The header handler also provides LAPI with an indication of the completion handler that must be executed when the entire message is copied into the target buffer specified by the header handler. The LAPI library moves the data (which may be transferred as multiple network packets) into the specified *buffer*. On completion of the data transfer the user−specified completion routine is invoked (Step 4). After the completion routine finishes execution, the *tgt_cntr* at the target process and *cmpl_cntr* at the origin process are updated indicating that the LAPI active message call is now complete. LAPI also provides a set of defined functions built on top of the active message infrastructure. These defined functions provide basic data transfer, synchronization, signaling, and ordering functions. LAPI can be used in either interrupt or polling mode. The typical mode of operation is expected to be interrupt mode. In the interrupt mode, a target process does not have to make any LAPI calls to assure communication progress. Polling mode may be used to provide better performance by avoiding the cost of interrupts. However a user of polling mode should be aware that in the absence of appropriate polling, the performance may substantially degrade or may even result in deadlock. The active message infrastructure and most LAPI functions built on top of that are non−blocking. The non−blocking nature allows the LAPI user to have a task initiate several concurrent operations to one or more target tasks. In our implementation, these concurrent calls return as soon as the messages has been queued at the

network, and do not have to wait for the communication event to actually complete. This "unordered pipelining" architecture results in significantly reducing (hiding) the per−operation latency.

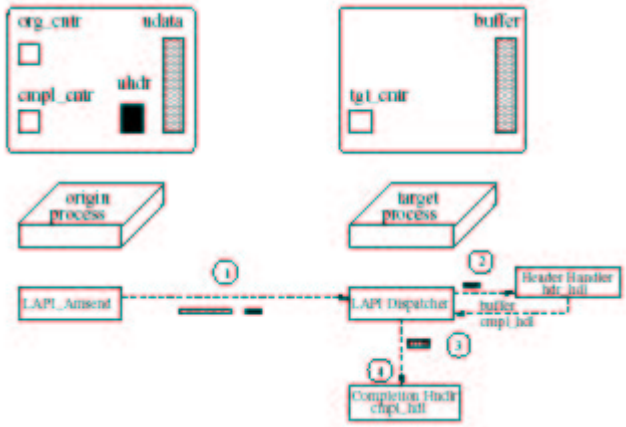


Fig. 1 − LAPI Overview

LAPI provides get and put operations to allow basic data copy from the address space of one process to that of another. They are sometimes referred to as remote memory copy (RMC) operations. Put copies data from the address space of the origin process to the address space of the target process; Get pulls data from the target process and copies it into the origin process. These operations are semantically unilateral or one−sided. The get or put is initiated by the origin process, and no complementary action by the target process is necessary for the call to complete. This is unlike traditional send/receive semantics, where a send has to be matched at the other end with a corresponding receive being posted with matching parameters before the data transfer operation can complete. Since get and put are unilateral operations, nonblocking, and not guaranteed to complete in order, the user is responsible for explicit process synchronization when necessary for program correctness.

Put, *Get* and *AM* are unilateral communication operations that are initiated by one task (the origin), but an indication of the completion of a communication operation is provided at both ends. The definition of when a *put* or *get* operation is complete needs some discussion. Intuitively, the origin may consider a *put* as complete when the data has been moved from the origin to the target, (i.e., The data is available at the target, and the origin data may be changed). However, another equally valid interpretation is one where the origin task considers the operation to be complete when the data has been copied out of its buffer and either the data is safely stored away or is on their way to the target. The target task would consider the *put* complete, when the data has arrived into the target buffer. Similarly, for a *get*, the target task may consider the operation to be complete when the data has been copied out of the target buffer, but has not yet been sent to the origin task. In order to provide the ability to exploit these different intuitive notions, LAPI has a completion notification mechanism via the use of counters. The user is responsible for associating counters with events related to message progress. However, the counter structure is an opaque object internally defined by LAPI and the user is expected to access the counter using only
the appropriate interfaces provided in LAPI. The user may use the same counter across multiple messages. This gives the user the freedom to group different communication calls with the same counter and check their completion as a group. The LAPI library updates the user specified counters when a particular event (or one of the events) with which the counter was associated has occurred. The user can either periodically check the counter value (using the non−blocking polling LAPI function *Getcntr*) or can wait until the counter reaches a specified value (using the blocking LAPI *Waitcntr* function). On return from the *Waitcntr* call, the counter value is automatically decremented by the value specified in the *Waitcntr* call.

LAPI operations decouple synchronization from data movement and there is no need for bilateral coordination of data transfers between the origin and target. For maximum performance concurrent operations may complete out of order. As a result, data dependencies between the source and the destination must be enforced using explicit synchronization as is the case in the shared memory programming style. However, in many cases the program structure makes it unnecessary to synchronize on each data transfer. LAPI provides atomic operations for synchronization.

Two LAPI operations that have the same origin task, are considered to be ordered with respect to the origin if one of the operations starts after the other has completed at the origin task. Similarly, two LAPI operations that have the same target task, are considered to be ordered with respect to that target, if one of the operations starts after the other has completed at the target task. If two operations are not ordered they are *concurrent*. LAPI provides no guarantees of ordering for concurrent communication operations. For example, consider the case where a node issues two non−blocking *puts* to the same target node, where the target buffers overlap. These two operations may complete in any order, including the possibility of the first *put* partially overlapping the second, in time. Therefore, the contents of the overlapping region will be undefined, even after both the *puts* complete. Waiting for the first to complete (for instance using the completion counter) before starting the second, will ensure that the overlapping region contains the result of the second, after both *puts* have completed. Alternatively, a *fence* call can be used to enforce order.

PB

PM communication library was proposed with Score, a Unix−like operating system, to provide a multi−user parallel environment on cluster of workstations. PM provides multiple channels so a user process can share the NI with the Score kernel communication tasks. PM also time−multiplexes each channel among different processes, using primitives to save and restore a channel state.

PM uses a new flow control algorithm called modified ACK/NACK to achieve reliable in−order delivery and yet be scalable with respect to the number of workstations. PM provides intra−message pipelining during a message send operation and inter−message pipelining on both send and receive operations. PM adopts polling for message arrival notification but also permits interrupts. PM 1.2 enables zero−copy transfers via remote write operations and a TLB placed in NI memory. Only 1024 pages can be pinned to memory with PM. This memory is dynamically treated as a cache by user processes, the pin−down cache. The user keeps a list of pages previously pinned−down, so as to avoid trying to pin pages that are already pinned−down.

DBM

The distinguishing feature of BDM is its ability to provide five different levels of service to the user: unreliable service, where packets may be lost because of software or hardware errors; bounce/unordered service, where packets may be lost because of hardware errors, return−to−sender flow control is used to cope with software errors, and packets may be delivered out−of−order; bounce/ordered service, where packets may be lost because of hardware errors and return−to−sender flow control together with a sequence number is used to cope with software errors and to preserve in−order deliver; reliable/unordered service, which implements ACK/NACK flow control and CRC checks to guarantee reliable out−of−order delivery; and reliable/ordered service, which implements ACK/NACK flow control, CRC checks, and sequence numbers to guarantee reliable in−order delivery.

The best performance is achieved by the unreliable service. The reliable protocols are not efficient, but can recover from hardware and software errors. Hardware errors include transmission bit errors, unplugged cables or a network switch turned off, while software errors are due to lack of buffer space on the receiving side. BDM is implemented through descriptor queues that manage buffers allocated in the NI memory. It is up to the host to move data to and from the NI, that is, BDM does not use the NI DMA to move data to the network. Notification of message arrival is done by polling.

Trapeze

The Trapeze messaging system consists of two components: a messaging library that is linked into the kernel or user programs, and a firmware program that runs on the Myrinet network interface card (NIC). The Trapeze firmware and the host interact by exchanging commands and data through a block of memory on the NIC, which is addressable in the host's physical address space using programmed I/O. The firmware defines the interface between the host CPU and the network device; it interprets commands issued by the host and controls the movement of data between the host and the network link. The host accesses the network using macros and procedures in the Trapeze library, which defines the lowest level API for network communication across the Myrinet. Since Myrinet firmware is customer−loadable, any Myrinet site
can use Trapeze.

Trapeze was designed primarily to support fast kernel−to−kernel messaging alongside conventional TCP/IP networking. Trapeze currently hosts kernel−to−kernel RPC communications and zero−copy page migration traffic for network memory and network storage, a user−level communications layer for MPI and distributed shared memory, a low−overhead kernel logging and profiling system, and TCP/IP device drivers for FreeBSD and Digital UNIX. These drivers allow a native TCP/IP protocol stack to use a Trapeze network through the standard BSD *ifnet* network driver interface.

Trapeze messages are short *control messages* (maximum 128 bytes) with optional attached *payloads* typically containing application data not interpreted by the networking system, e.g., file blocks, virtual memory pages, or a

TCP segments. The data structures in NIC memory include two message rings, one for sending and one for receiving. Each message ring is a circular array of 128−byte control message buffers and related state, managed as a producer/consumer queue shared with the host. From the perspective of a host CPU, the NIC produces incoming messages in the receive ring and consumes outgoing messages in the send ring.

Trapeze has several features useful for high−speed TCP/IP networking: separation of header and payload, large MTUs with scatter/gather DMA and adaptive message pipelining.

One item missing from this list is *interrupt suppression.* Handling of incoming messages is interrupt−driven when Trapeze is used from within the kernel; incoming messages are routed to the destination kernel module (e.g., the TCP/IP network driver) by a common interrupt handler in the Trapeze message library. Interrupt handling imposes a per−packet cost that becomes significant with smaller MTUs. Some high−speed network interfaces reduce interrupt overhead by amortizing interrupts over multiple packets during periods of high bandwidth demand. For example, the Alteon Gigabit Ethernet NIC includes support for adaptive interrupt suppression, selectively delaying packet−receive interrupts if more packets are pending delivery. Trapeze implements interrupt suppression for a lightweight kernel−kernel RPC protocol, but we do not use receive−side interrupt suppression for TCP/IP because it yields little benefit for MTUs larger than 16KB at current link speeds.

MyriAPI
MyriAPI is Myricom's MSA for the Myrinet. It supports multi−channel communication, scatter−gather operations, and dynamic network configuration. It does not provide reliable message delivery. Buffers are allocated in a statistically pinned region at the host memory. It uses three descriptor queues (free, send and receive) of scatter−gather message pointers that are shared by the host and the NI. Data are always transferred to and from the NI via DMA operations. Notification is done by polling.

Virtual Interface Architecture (VIA)
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. 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.

VI ARCHITECTURE OVERVIEW

The Virtual Interface Architecture represents a significant deviation from traditional OS−network interfaces, placing more direct access to the network in the user space while attempting to provide the same protection as that provided by operating system controlled protocol stacks. The VI Architecture provides hardware support for direct user access to a set of virtual network interfaces, typically without any need for kernel intervention.

The VI Architecture is designed to address three important problems associated with the high cost of network access:

- Low bandwidth − Network−related software overhead limits the usable bandwidth of the network. In many instances only a fraction of the possible network bandwidth can be utilized.
- Small message latency − Because processes in a distributed system must synchronize using network messages, high latency for typically small synchronization messages can greatly reduce overall performance.

- Processing requirements − The overhead of message processing can significantly reduce processing time that the CPU can dedicate to application code.

The VI Architecture is designed to reduce the amount of processing overhead of traditional network protocols by removing the necessity of a process taking a kernel trap on every network call. Instead, consumer processes are provided a direct, protected interface to the network that does not require kernel operations to send or receive messages. Each such virtual interface is analogous to the socket endpoint of a traditional TCP connection: each VI is bidirectional and supports point−to−point data transfer. Support for these virtual interfaces, are intended to be implemented in hardware on the network interface card (NIC). The network adapter performs the endpoint virtualization directly and performs the tasks of multiplexing, de−multiplexing, and data transfer scheduling normally performed by an OS kernel and device driver. At the time of writing, the majority of VI implementations do not support these functions in hardware. Instead, standard NIC's are used, and the necessary VI support is emulated by device driver software.

Fig. 2 depicts the organization of the Virtual Interface Architecture. The VI Architecture is comprised of four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers. The VI Provider consists of the VI network adapter and a Kernel Agent device driver. The VI Consumer is generally composed of an application program and an operating system communication facility such as MPI or sockets, although some "VI−aware" applications communicate directly with the VI Provider API. After connection setup by the Kernel Agent, all network actions occur without kernel intervention, resulting in significantly lower latencies than network protocols such as TCP/IP. Traps into kernel mode are only required for the creation and destruction of VI's, VI connection setup and tear−down, interrupt processing, registration of system memory used by the VI NIC, and error handling. VI Consumers access the Kernel Agent using standard operating system mechanisms.

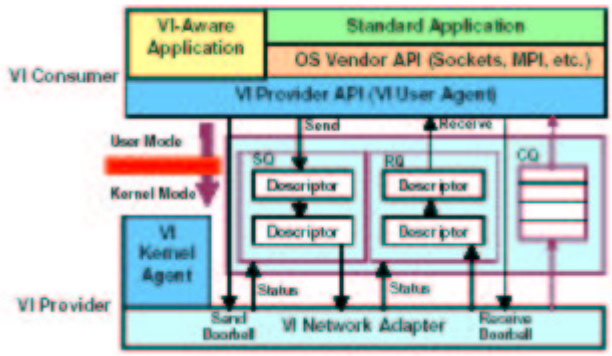


Fig. 2 − Organisation of the Virtual Interface Architecture

A VI consists of a Send Queue and a Receive Queue. VI Consumers post requests (Descriptors) on these queues to send or receive data. Descriptors contain all of the information that the VI Provider needs to process the request, including pointers to data buffers. VI Providers asynchronously process the posted Descriptors and mark them when completed. VI Consumers remove completed Descriptors from the Send and Receive Queues and reuse them for subsequent requests. Both the Send and Receive Queues have an associated "Doorbell" that is used to notify the VI network adapter that a new Descriptor has been posted to either the Send or Receive Queue. In a hardware VI Architecture implementation, the Doorbell is directly implemented on the VI Network Adapter and no kernel intervention is required to perform this signaling. The Completion Queue allows the VI Consumer to combine the notification of Descriptor completions of multiple VI's without requiring an interrupt or kernel call.

Connection Procedures

The VI Architecture provides a connection−oriented networking protocol similar to TCP. Programmers make operating system calls to the Kernel Agent to create a VI on the local system and to connect it to a VI on a remote system. Once a connection is established, the application send and receive requests are posted directly to the local VI. A process may open multiple VI's between itself and other processes, with each connection transferring information between the hosts. The VI architecture provides both reliable and unreliable delivery connections. The expense of setting up and tearing down VI's means that connections are typically made at the beginning of program execution.

Memory Registration

The VI architecture requires the VI Consumer to register all send and receive memory buffers with the VI Provider in order to eliminate the copying between kernel and user buffers. This copying typically accounts for a large portion of the overhead associated with traditional network protocol stacks. The registration process locks the

appropriate pages in memory, allowing for direct DMA operations into user memory by the VI hardware without the possibility of an intervening page fault. After locking the buffer memory pages in physical memory, the virtual to physical mapping and an opaque handle for each memory region registered are provided to the VI Adapter. Memory registration allows the VI Consumer to reuse registered memory buffers and avoid duplicate locking and translation operations. Memory registration also takes page−locking overhead out of the performance−critical data transfer path. Since memory registration is a relatively expensive VI event, it is usually performed once at the beginning of execution for each buffer region.

Data Transfer Modes

The VI Architecture provides two different modes of data transfer: traditional send/receive semantics, and direct reads from and writes to the memory of remote machines. Remote data reads and writes provide a mechanism for a process to send data to another node or retrieve data from another node, without any action on the part of the remote process (other than VI connection). The send/receive model of the VI Architecture follows the common approach to transferring data between two endpoints, except that all send and receive operations complete asynchronously. Data transfer completion can be discovered through one of two mechanisms: polling, in which a process continually checks the head of the Descriptor Queue for a completed message; or blocking, in which a user process is signaled that a completed message is available using an operating system synchronization object.

Myrinet Network

Myrinet is a cost−effective, high−performance, packet−communication and switching technology that is widely used to interconnect clusters of workstations, PCs, servers, or single−board computers. Clusters provide an economical way of achieving high performance**,** by distributing demanding computations across an array of cost−effective hosts and high availability**,** by allowing a computation to proceed with a subset of the hosts.

Conventional networks such as Ethernet can be used to build clusters, but do not provide the performance or features required for high−performance or high−availability clustering. Characteristics that distinguish Myrinet from other networks include:

- Full−duplex 2+2 Gigabit/second data rate links, switch ports, and interface ports.
- Flow control, error control, and "heartbeat" continuity monitoring on every link.
- Low−latency, cut−through, crossbar switches, with monitoring for high−availability applications.
- Switch networks that can scale to tens of thousands of hosts, and that can also provide alternative communication paths between hosts.
- Host interfaces that execute a control program to interact directly with host processes ("OS bypass") for low−latency communication, and directly with the network to send, receive, and buffer packets

Myrinet, developed by Myricom, uses variable−length packets. The packets are wormhole−routed through a network of highly reliable links and crossbar switches. Techniques like wormhole routing are normally found only in supercomputers. In wormhole networks, packets are comprised of small units called flits. The key distinction from store−and−forward network is that a flit is forwarded to the output port (if its is free) as soon as possible, without waiting for the entire packet to arrive at the input port. This allows lower latency and better network utilisation. Fig. 3 illustrates the architecture of a node in a Myrinet cluster.

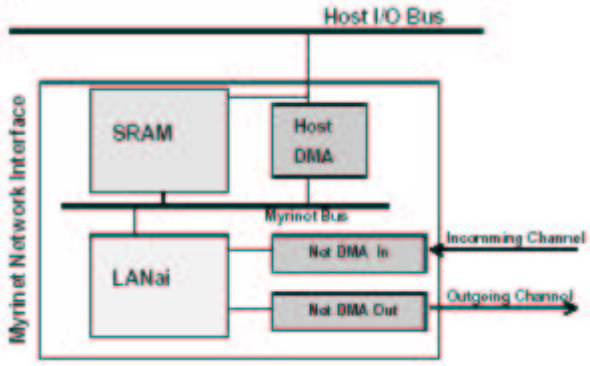


Fig. 3 − The Host and Network Interface Architecture of Myricom's Myrinet

Myrinet is a high−speed local−area network or system area network for computer systems. A Myrinet network is composed of point−to−point links connecting hosts and/or switches. Each network link can deliver a maximum of 1.28 Gbits/s bandwidth in each direction.

The Myrinet network provides in−order network delivery with low bit error rates (below 10−15). On sending, an 8−bit CRC is computed by hardware and is appended to the packet. On packet arrival, the CRC of each packet is computed anew by dedicated hardware and is compared with the received CRC. If they do not match, a CRC error is reported.

The Myrinet PCI−bus network interface card contains a 32−bit control processor called LANai with 1 MByte of SRAM 3. The SRAM contains network buffers in addition to all code and data for the LANai processor. There are three DMA engines on the network interface: two for data transfers between the network and SRAM, and one for moving data between the SRAM and the host main memory over the PCI bus. The DMA engine that transfers data between the network interface and the host memory uses physical addresses to access host memory. The LANai processor is clocked at 33 MHz and executes a LANai control program (LCP), which supervises the operation of the DMA engines and implements a lowlevel communication protocol. The LANai processor cannot access the host memory directly; instead it must use the host−to−LANai DMA engine. The internal bus clock runs at twice the CPU clock speed, allowing up to two DMA engines to operate concurrently at full speed. The host can also access both the SRAM and the network itself, by using Programmed I/O (PIO) instructions. All or part of the network interface resources (memory and control registers) can be mapped to user memory.

Emerging Virtual Interface Applications

The performance of high−speed network storage systems is often limited by client overhead, such as memory copying, network access costs and protocol overhead. A related source of inefficiency stems from poor integration of applications and file system services; lack of control over kernel policies leads to problems such as double caching, false prefetching and poor concurrency management. As a result, databases and other performance−critical applications often bypass file systems in favor of raw block storage access. This sacrifices the benefits of the file system model, including ease of administration and safe sharing of resources and data. These problems have also motivated the design of radical operating system structures to allow application control over resource management.

The recent emergence of commercial direct−access transport networks creates an opportunity to address these issues without changing operating systems in common use. These networks incorporate two defining features: user−level networking and remote direct memory access (RDMA). User−level networking allows safe network communication directly from user−mode applications, removing the kernel from the critical I/O path. RDMA allows the network adapter to reduce copy overhead by accessing application buffers directly.

Network storage solutions can be categorized as Storage−Area Network (SAN)−based solutions, which provide a block abstraction to clients, and Network−Attached Storage (NAS)−based solutions, which export a network file system interface. Because a SAN storage volume appears as a local disk, the client has full control over the volume's data layout; client−side file systems or database software can run unmodified. However, this precludes concurrent access to the shared volume from other clients, unless the client software is extended to coordinate its accesses with other clients. In contrast, a NAS−based file service can control sharing and access for individual files on a shared volume. This approach allows safe data sharing across diverse clients and applications.

Communication overhead was a key factor driving acceptance of Fibre Channel as a high−performance SAN. Fibre Channel leverages network interface controller (NIC) support to offload transport processing from the host and access I/O blocks in host memory directly without copying. Recently, NICs supporting the emerging iSCSI block storage standard have entered the market as an IP−based SAN alternative. In contrast, NAS solutions have typically used IP−based protocols over conventional NICs, and have paid a performance penalty. The most−often cited causes for poor performance of network file systems are (a) protocol processing in network stacks; (b) memory copies; and (c) other kernel overhead such as system calls and context switches. Data copying, in particular, incurs substantial per−byte overhead in the CPU and memory system that is not masked by advancing processor technology.

The benefits of low client overhead and zero copy technology are gathering support for using VI throughout the data access path. A framework for understanding how applications access data is illustrated in Fig. 4. Most applications use the services provided by a file/record subsystem to access data. This subsystem in turn uses the services of a

block subsystem to access data stored as blocks on physical media. Functions in both subsystems can be performed in the host, the network, the storage device, or combinations thereof.

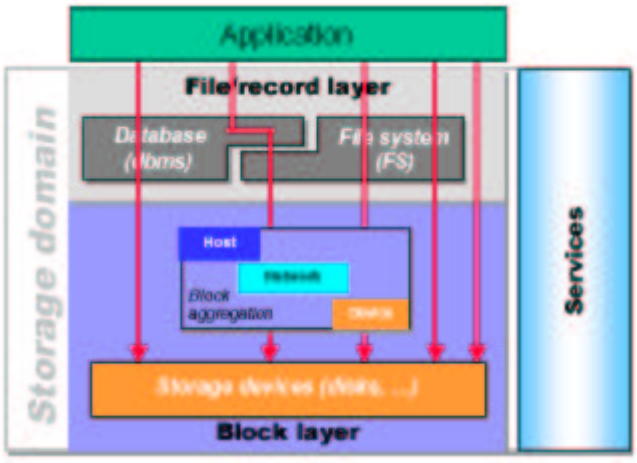
Fig. 4 − The SNIA Storage Model

File systems themselves may be divided into client and server components and distributed across a network. The advantage of this approach is that multiple clients, running on different host operating systems, can share files by using the same file server. The tradeoff of this approach has been a relative performance penalty associated with client/server communication over a network. VI and the related communication mechanisms specified by InfiniBand verb behavior, provide an innovative way to not only speed client/server communication, but to return more application server resources to the application.

The Direct Access File System (DAFS) is a new, fast and lightweight file access protocol that uses the underlying Vi architecture capabilities as its standard transport mechanism to provide direct application access to shared file servers. It is optimized for high−performance (high throughput, low−latency communication), mission−critical data centre environments and for the requirements of local file−sharing architectures. Local file sharing requires high−performance file and record locking in order to maintain consistency of shared data. DAFS allows locks to be cached so that repeated access to the same data need not result in a file server interaction. Should a node require a lock cached by another node, the lock is transferred at network speeds without requiring timeouts. DAFS is also designed to be resilient in the face of both client and file server reboots and failures.

Given the nature of VI (networked blocks of shared memory), the VI architecture and DAFS require some level of trust between clustered machines. However, DAFS will provide secure user authentication so that access to information can be controlled at a high level. Additionally, the servers comprising a cluster maintain a table of their partner servers. New servers cannot enter the cluster to share data unless permitted by the existing members. Existing servers ejected from the cluster are prevented from accessing data that was formerly authorized.

Based on NFSv4, DAFS is enhanced to take advantage of standard memory−to−memory interconnects. DAFS supports client implementations suitable for database and operating system vendors as well as native DAFS applications.



The DAFS enables a new paradigm for shared data access that goes beyond the current capabilities of network attached storage, direct−attached storage, and Fibre Channel SAN architectures. DAFS is specifically designed to enhance local file sharing architectures, which in turn enables a platform to deploy scalable and reliable services based on cost−effective commodity server hardware. It was introduced to combine the low overhead and flexibility of SAN products with the generality of NAS file services.

Other developments focus on mapping traditional protocols over Remote DMA networks. Microsoft Windows 2000 Datacenter Server allows applications to automatically leverage VI networks using its Winsock Direct interface. The Direct Access Sockets (DASockets) open source standardization effort is also working to provide interoperable traditional sockets emulation on other open system platforms. And a SCSI RDMA Protocol (SRP) project in the ANSI T10 Technical Committee is standardizing a SCSI protocol mapping onto cluster protocols such as InfiniBand.

The DASockets protocol defines the on−the−wire data transport protocol between nodes. The protocol requires a VI like interface as the abstraction to interconnect hardware. Thus it is natural for DASockets to operate most efficiently on top of memory−to−memory interconnect (MMI) SAN fabrics, and it is so designed.

The DASockets protocol does not define nor constrain the interconnect implementation. It merely requires that communication be defined through logical channels that, ideally, map directly to Transport level provider channels (*e.g.*, connected VIs). These channels (called Transport Resources) must provide reliable, in−order, exactly−once delivery of messages and RDMA write operations. No assumption is made about ordering of messages or RDMA operations across Transport Resources.

DASockets can emulate traditional TCP sockets (BSD 4.4 or Winsock2) behavior. No TCP/IP stack is required to transfer data between two endpoints, since the underlying interconnect provides the reliable transport. DASockets merely fills in the remaining TCP behavioral semantics. IP semantics (routing, broadcasting, etc.) are not modeled, but endpoints do retain IP addressing. No datagram service is provided out of simplicity, and the belief that the traditional low−overhead, low−latency rationale for datagrams is rendered largely moot.

The DASockets protocol is optimized so that an implementation of traditional TCP sockets can be efficiently layered above it. The protocol does not restrict, indeed it invites, socket API extensions for enhanced performance.

REFERENCES

Compaq Computer Corp., Intel Corporation, Microsoft Corporation. Virtual Interface Architecture Specification, Version 1.0. http://www.viarch.org/. December 16, 1997.

D. Dunning and G. Regnier. The virtual interface architecture. InHot Interconnects V, pages 47−−58, 1997.

P. Buonadonna, A. Geweke, and D. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In Supercomputing ’98, Orlando, FL, November 1998.

Dave Running, Greg Regnier, et al. The virtual interface architecture. IEEE Micro, 18(2):66−76, March/April 1998.

T. von Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. IEEE Computer, 31(11):61 −− 68, Nov. 1998.

W. E. Speight, H. Abdel−Shafi, and J. K. Bennett. Realizing the performance potential of the virtual interface architecture. In Proc. of the 13th ACM−SIGARCH Int’l Conf. on Supercomputing (ICS’99), June 1999.

Philip Buonadonna, Joshua Coates, Spencer Low, and David E. Culler. Millennium sort: A clusterbased application for windows nt using dcom, river primitives and the virtual interface architecture. In Proceedings of the 3rd USENIX Windows NT Symposium, July 1999.

F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. IEEE Micro, 18#2#:66#??, Mar.#Apr. 1998.

Infiniband Architecture Specification Volume 1, Release 1.0. Infiniband Trade Association, 2000.

Daniel Cassiday. Infiniband architecture tutorial. Hot Chips 12 Tutorial, August 2000.

William T. Futral. InfiniBand Architecture Development and Deployment. Intel, 2001.

D. Pendery and J Eunice. InfiniBand Architecture: Bridge Over Troubled Waters, Research Note, InfiniBand Trade Ass’n, April 27, 2000, available from www.infinibandta.com

G. Pfister, "An Introduction to the InfiniBand Arechitecture", High Performance Mass Storage and Parallel I/O , IEEE Press, 2001. www.infinibandta.org

J. Wu et al. " Design of An InfiniBand Emulation over Myrinet: Challenges, Implementation, and Performance Evaluation," Technical Report OUS−CISRC−2/01_TR−03, Dept. of Computer and Information Science, Ohio State University, 2001

Chris Eddington. InfiniBridge: An InfiniBand channel adapter with integrated switch. IEEE Micro, 22(2):48−−56, March/April 2002.