# CHAPTER III

# MESSAGE LAYERS FOR RESOURCE-RICH CLUSTER COMPUTERS

As discussed in the previous section, cluster computers provide a cost-effective platform for processing distributed applications. If a cluster computer's communication library is visualized as a means of providing a virtual parallel-processing machine for distributed applications, there is one component of the virtual architecture that current generation communication libraries omit: peripheral devices. Traditional cluster communication libraries are designed to transfer data only between host CPUs, not peripheral devices. For these libraries it is assumed that cluster interactions with a peripheral device are performed by a host-level application that resides in the same host as the device. Therefore in order to interact with a remote peripheral device, an application must communicate with the remote host's CPU and request an operation be performed on behalf of the application. This method of controlling a peripheral device through a proxy incurs costly overheads that limit the dynamic use of peripheral devices in distributed applications.

The fact that peripheral devices can strongly influence a cluster computer's performance challenges the notion that communication libraries should be designed in such a CPU-centric manner. Peripheral devices are becoming increasingly more powerful and therefore represent a valuable opportunity for accelerating cluster computer applications. The inclusion of powerful peripheral devices in the cluster architecture results in a new category of cluster computer that we refer to as *resource-rich cluster computers*. Since these clusters exhibit different communication requirements than traditional clusters, it is beneficial to examine the design of new communication libraries that are well suited to these clusters. These libraries specifically allow both host CPUs and peripheral devices to be efficiently utilized as computational resources by distributed applications. The resulting virtual parallel-processing machine provided by the communication library is depicted in Figure 3.1.

This chapter provides the groundwork for designing message layers that are well suited for resource-rich cluster computers. As a first step in this effort, definitions of the hardware

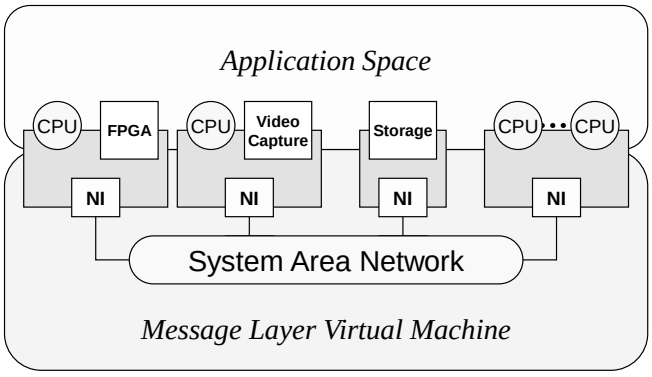


**Figure 3.1:** Including peripheral devices in the virtual parallel-processing machine architecture provided by the communication library.

environment are provided as well as motivating examples of how these clusters can be beneficial to end users. This work is followed by a discussion of fundamental concepts that influence the construction of the communication library. These concepts are then individually elaborated. Finally, this chapter is concludes with a listing of research projects that are related to resource-rich clusters.

## *3.1 Emergence of Resource-Rich Cluster Computers*

Resource-rich cluster computers are clusters where individual workstations are supplemented with powerful peripheral devices. It is important to examine the characteristics of these clusters in order to determine how message layers should be designed for these systems.

### 3.1.1 Availability of Powerful Peripheral Devices

In recent years commercial hardware vendors have constructed a number of powerful peripheral devices that are designed to perform a variety of application-specific operations. One of the key motivations for this effort has been the need to produce high-performance network servers for the Internet. In answer to market demand developers have constructed a number of intelligent I/O cards for both LAN and storage operations. For LANs, developers have constructed high-performance network cards that feature embedded processors and multiple physical links to the network. Some of these cards are equipped with firmware that allows common network operations such as TCP connection management to be performed on the network card as opposed to the host. Storage controller cards are also becoming increasingly more powerful due to the active disk [77] and network-attached storage (NAS) [65] efforts. Modern intelligent storage adaptor cards are capable of managing a disk's file system at the controller in a self-contained manner. These storage cards provide a file-level interface to end applications and do not require the assistance of the host's operating system.

Another area where peripheral devices are becoming more powerful is in multimedia applications. Driven by consumer interest in high-quality video and audio editing, developers have made significant improvements to multimedia peripheral devices. Modern audio and video capture devices can be configured to automatically push high-resolution data samples directly into host memory, allowing data streams to be captured in real time. Some of these cards feature hardware to perform desirable operations such as compression, clipping, and filtering. Other multimedia cards are available for rendering high-quality output for people to observe. Audio playback and video display cards generate output signals from large on-card buffers that can be written to by applications. Some of these output cards feature processing devices that are capable of performing significant computations in real time.

A third area where peripheral devices have become more powerful is in the field of computational accelerator cards. These cards are designed to utilize custom hardware to improve the performance of certain types of computations. Typically these cards employ digital signal processors (DSPs), field-programmable gate arrays (FPGAs), or even dedicated application-specific integrated circuits (ASICs). Often these cards feature large amounts of high-speed memory for storing large data sets on the card and function as a form of co-processor for the host. The common procedure for utilizing a hardware accelerator card is for the host application to pass data to the card, have the peripheral device process the

data, and then have the results transferred back to the host. Custom hardware accelerator cards are often useful for processing large streams of data such as multimedia traffic.

### 3.1.2 Categorizing Peripheral Devices

Based on the previous examples it is possible to broadly categorize peripheral devices by the manner in which they are utilized. Three common categories include the following.

- **Data Sources and Sinks**: Peripheral devices are often utilized to produce data for the host (i.e., a data source) or store data from the host (i.e., a data sink). Some peripheral devices such as storage adaptor cards are capable of performing both of these operations. Typically these devices do not perform elaborate computations on incoming or outgoing data.
- **Intermediate Processing Elements**: Peripheral devices such as the custom hardware accelerators are primarily designed to process data for the host. Data is typically injected into these cards, processed, and then ejected back to the host system. Incoming and outgoing data rates for these cards do not have to be equal and are dependent on the application.
- **System Bridges**: A peripheral device can also be designed to serve as a form of bridge between two separate systems. The bridge device therefore manages communication between the two systems, performing protocol translations when needed. One example of a bridge is a LAN adaptor card that is utilized to connect the cluster to an application running at a host that is not part of the cluster. The cluster side of the bridge communicates with a SAN protocol while the external side utilizes a LAN protocol (such as TCP).

### 3.1.3 Characteristics of Resource-Rich Cluster Computer Hardware

A resource-rich cluster's hardware architecture is similar to traditional cluster computers, with the exception that workstations are equipped with powerful peripheral devices. Physically adding these devices to the cluster is relatively simple, as cards are placed in the available PCI slots of a cluster's workstations. Figure 3.2 depicts the physical architecture of a resource-rich cluster computer. Each workstation features various peripheral devices and is connected to the cluster through a high-performance SAN. This SAN functions as a backbone for communication in the cluster and can be accessed by both host CPUs and peripheral devices.

While resource-rich clusters are not a radical departure from traditional cluster architectures, there are several unique characteristics that communication library designers must be aware of. The more significant characteristics include the following.

- **Two-levels of Communication Infrastructure**: Communication within a resource-rich cluster takes place in two distinct levels: in the local host context (intra-host) and between hosts using the SAN (inter-host). Intra-host communication can be facilitated with software that intelligently utilizes the host's local I/O system. Inter-host communication requires software that transfers data through both the local I/O system and backbone SAN substrate.

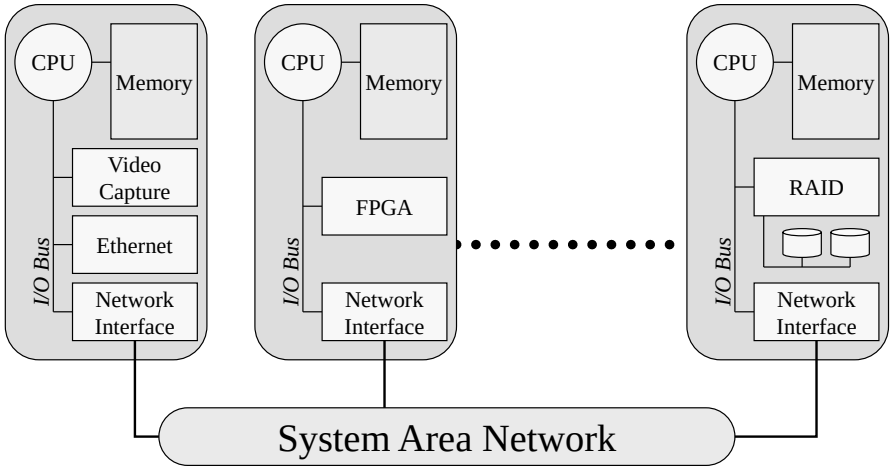


**Figure 3.2:** The inclusion of peripheral devices in the resource-rich cluster architecture.

- **Globally-Shared Peripheral Devices**: Resource-rich clusters feature a number of peripheral devices that can be utilized by end applications. While each device in the cluster is owned and managed by the operating system of the workstation in which it resides, it is beneficial for devices to be accessible in the global context of the cluster. The ultimate goal is for any resource to be able to efficiently utilize any other resource in the cluster.

- **Differences in Peripheral Device Capabilities**: Peripheral devices are generally designed to perform specific functions using minimal amounts of hardware resources. While some devices feature programmable embedded processors and large amounts of on-card memory, others may only be equipped with low-speed ASICs configured with simple state machines. Therefore different peripheral devices have different capabilities. These differences influence the extent to which a device can be integrated into the resource-rich cluster environment and made available as a global resource.

- **Limited Local I/O Capacity**: Workstations have a fixed capacity for local I/O operations. In addition to being limited, local I/O bandwidth is generally shared among all peripheral devices in a host. Therefore it is important that data transfers involving the local I/O system be orchestrated in an efficient manner. For example, if data is being moved from one resource to another in a host, it should be transferred directly with a single copy as opposed to a two-copy approach where the data is first transferred into an intermediate host buffer.

### 3.1.4 Resource-Rich Cluster Computer Applications

There are a number of applications that can benefit from the use of resource-rich cluster computers. One motivating example can be found in the field of high-performance network servers. As depicted in Figure 3.3, a resource-rich cluster could be used to implement a tightly synchronized web server that is capable of sustaining high network loads. In this example each host in the cluster utilizes an intelligent LAN adaptor card to service incoming requests from external clients and an intelligent storage adaptor to house portions of a large database. In order to service incoming requests, the LAN adaptors communicate directly

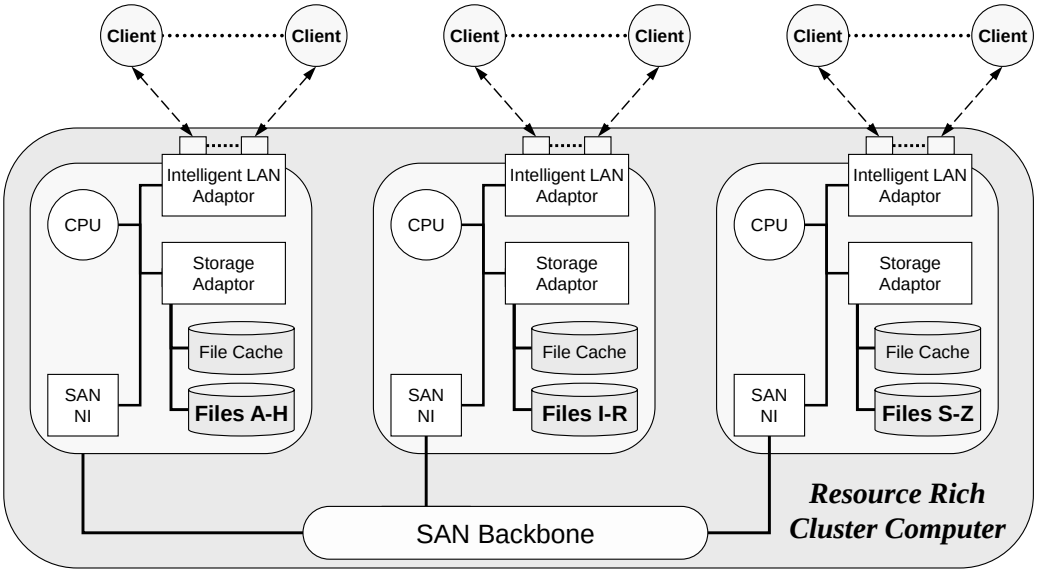


**Figure 3.3:** An example of a resource-rich cluster functioning as a network server.

with the appropriate storage controller card using the SAN and the communication library. This form of large-scale server is particularly useful for applications such as digital libraries, where the database is enormous and cannot simply be replicated at each host.

Resource-rich cluster computers can also be utilized for applications that process large multimedia data streams. In a full-scale multimedia task, audio and video data is acquired by multimedia capture devices, streamed through various computational resources in the cluster, and then ejected to either storage or output devices. In this application the communication library must efficiently transfer data between cluster resources in order to meet real-time requirements. A variant of this task is to utilize host CPUs to generate the data streams instead of capture devices. An example of this task is illustrated in the WireGL project [43], where multiple hosts in a cluster generate objects that are combined and rendered to a grid of output displays. These types of operations can be beneficial in scientific applications where a small cluster is utilized to graphically render the computational results of a larger cluster [26, 76].

## *3.2 Design of Message Layers for Resource-Rich Clusters*

Physically constructing a resource-rich cluster is a relatively straightforward task: individual components of the architecture can be purchased and assembled from commodity parts that are widely available. A more challenging task is constructing software that allows the hardware to function as part of a single system. Utilizing commodity software such as the open source GNU/Linux operating system is a significant first step in this effort. Linux provides well-defined APIs and built-in device drivers for managing many different hardware devices. However, current generation commodity operating systems are only designed to control a local host, not a cluster of hosts. What is needed is a communication library that is located in or directly above the operating system to provide an application with a means of utilizing the resources that are distributed throughout the cluster. As discussed in the previous chapter, this communication library serves as a means of presenting end users with

a form of virtual parallel-processing machine for distributed applications.

Existing communication libraries are inappropriate for resource-rich clusters because they do not provide mechanisms for accessing peripheral devices in the global context. Extending these communication libraries to provide such access is nontrivial or impossible because the libraries are optimized under the assumption that the NI is controlled exclusively by the host CPU. Therefore it is necessary to consider how a communication library can be designed with fundamentals that support the needs of resource-rich cluster computers. In this effort it is beneficial to examine both system level issues as well as practical features that assist end users. These factors influence the design of the communication library and must be addressed in order for a resource-rich cluster to function efficiently.

### 3.2.1 Definition of a Communication Endpoint

One of the first tasks in designing a message layer is defining what constitutes an endpoint in the cluster. In this thesis, a communication endpoint is a programming abstraction that allows a resource in the cluster to be connected to the message layer. This abstraction enables the resource to send messages to and receive messages from other resources that are available in the cluster. For resource-rich clusters, host CPUs and peripheral devices are allowed to function as communication endpoints.

There are three components of a general communication endpoint implementation. First, an endpoint utilizes a block of its local memory to serve as a place for housing queues for incoming messages. These queues allow other resources in the local host (e.g., the NI and other local endpoints) to pass messages directly to the endpoint's address space. Second, an endpoint is equipped with methods for interpreting and processing messages from the incoming message queues. These methods allow the endpoint to react to the stimulus of a new message. Finally, an endpoint features mechanisms for ejecting an outgoing message to another resource in the local host (e.g., the NI or a local endpoint). These mechanisms allow the endpoint to interact with other resources in the cluster.

While desirable, it is not necessary for a communication endpoint to implement all three of these message-passing components. Designers may omit one or more of these components based on the characteristics of the resource. For example, a peripheral device that functions as a data source only needs to be equipped with mechanisms for ejecting outgoing messages. Likewise, a data sink only needs to be able to accept and process incoming messages. The advantage of implementing all three components of the communication endpoint software is that doing so allows users to better customize their interactions with the resource. Users can send data requests to these resources and receive feedback or data results.

### 3.2.2 Architectural Design Issues

The architectural characteristics of a resource-rich cluster have a strong influence on the way that communication library software should be designed for these clusters. Key issues that must be addressed include the following.

- **End-to-End Flow Control**: Flow control is utilized as a means of preserving buffer space in the communication library implementation. Resource-rich clusters typically employ a large number of communication endpoints, many of which have limited computational facilities. Therefore it is infeasible for each endpoint to manage end-to-end flow control for delivering messages. Instead, resource-rich cluster communication

libraries should utilize per-hop flow control schemes that simplify the workload of the endpoints.

- **Shared NI Access in a Host**: In resource-rich clusters a host may be equipped with multiple communication endpoints at both the host CPU and peripheral device levels. Each of these endpoints must access the NI to communicate with other endpoints in the cluster. Therefore the communication library must provide efficient means of sharing the NI among multiple endpoints. These mechanisms must allow multiple endpoints to coherently inject data into the NI. For this task we propose the use of NI-based logical channels.

- **Flexible and Powerful Programming Model**: The communication library must provide a programming model that is flexible enough to serve the diverse needs of cluster users. This programming model must be able to support traditional host-to-host communication mechanisms as well as means of interacting with peripheral devices in the cluster. The model must also be extensible, allowing new functionality to be added by end users when necessary. We propose the use of two APIs in the communication library: one for active messages and the other for remote memory operations.

- **Simple Standardized Endpoint Interface**: A variety of diverse cluster resources must implement communication endpoint software. For robustness and portability it is useful if the endpoint interface adheres to a standard form that is universal for all endpoints. Since there is a large amount of diversity in the capabilities of peripheral devices, it is important that this interface be designed in a manner that allows it to be implemented on even the simplest of peripheral devices.

- **Optimizations**: Modern communication libraries are expected to deliver high levels of performance for traditional host-level transactions. While a communication library for a resource-rich cluster trades some performance for increased functionality, the library should still be able to provide reasonable amounts of host-level performance. Therefore it is necessary to include optimizations in the library when possible for improving performance.

### 3.2.3 Design Overview

Designing a communication library for a resource-rich cluster requires the construction of appropriate mechanisms to address the preceding design issues. While there are certainly many possible solutions, we define a list of four key design characteristics that can be utilized to provide a suitable communication library. These characteristics are discussed in detail in the following sections and summarized as follows. First, per-hop flow control can be utilized to address the need for dynamic buffer management in the communication library without complicating the communication endpoint software. Second, the use of multiple logical channels in the NI allows communication endpoints in a host to share a NI without heavyweight synchronization protocols. Third, an active message style programming interface provides a uniform means by which end users can efficiently utilize peripheral devices. Finally, the programming interface can be supplemented with methods for interacting with remote endpoint memory in order to improve the flexibility of the library as well as its performance.

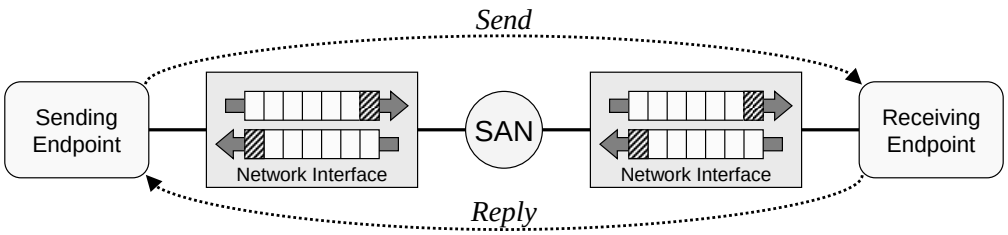


**Figure 3.4:** Endpoint-managed flow control schemes typically require send/reply messages to be transferred between endpoints to manage flow control credits. The dark buffers in the NIs represent buffer space that is reserved until the send/reply transaction completes.

## *3.3 Per-hop Flow Control*

Reliable communication libraries utilize flow control mechanisms to manage buffer space in the library. Without flow control an incoming message can erroneously overwrite an in-flight message that has not yet been processed. For simplicity several reliable message layers implement flow control at the host level. This approach can be labeled as endpoint-managed flow control, and requires an endpoint to acquire a flow control credit for the intended destination before it injects a message into the NI. The credit represents a reservation of buffer space along the entire communication pathway in the library (i.e., the sending and receiving NIs and the receiving endpoint). Endpoints must maintain flow control state information and communicate with other endpoints when updating this information.

Endpoint-managed flow control is inappropriate for resource-rich clusters because it complicates endpoint responsibilities. A more appropriate mechanism recommended for resource-rich clusters is to manage flow control on a per-hop basis. In this approach a message can progress along its communication pathway when enough buffer space is available to receive the message in the next communication stage. While this adds complexity to the design of the communication library it simplifies the work a communication endpoint must perform in order to interact with the library. A key element of this design is managing flow control between NI pairs. An optimistic approach is suggested for this effort in order to reduce communication latency.

### 3.3.1 Disadvantages of Endpoint-Managed Flow Control

In an endpoint-managed flow control schemes an endpoint must secure a reservation of buffer space for a message from all of the network elements that will be used to transfer the message before the message can be injected into the network. Rather than perform reservations on-demand, most endpoint-managed flow control schemes use a credit-based reservation system, where network buffers are allocated in advance and assigned to the endpoints in the system. An endpoint has a limited number of credits to communicate with each endpoint in the system and must spend a credit before the communication can begin. After receiving a message an endpoint must transmit a credit-replenishing reply to the sender. An example of this scheme for a single transaction is depicted in Figure 3.4. The shaded regions in the message queues represent buffer space that is allocated for a transmission during the time between when the message is first transmitted and the reply is received.

There are several negative aspects of endpoint-managed flow control for both traditional and resource-rich clusters. First, endpoint-managed flow control schemes perform injection

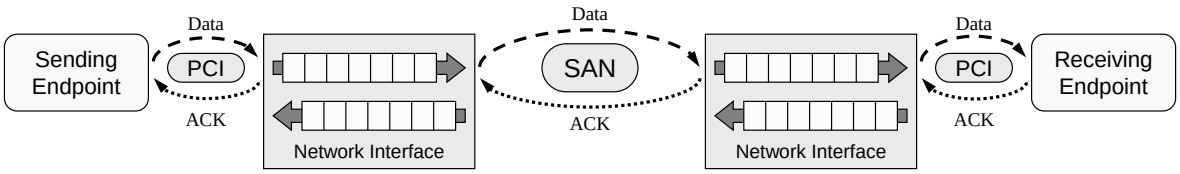


**Figure 3.5:** Per-hop flow control utilizes synchronization in the communication path to allow messages to progress when buffer space is available.

policing at a coarse granularity. It is possible that an endpoint will delay injecting a message into a NI that has buffer space for the message, simply because buffer space has not yet been reserved for the entire communication path. Second, endpoint-managed flow control schemes require credit information to flow between endpoints. This information adds to the network traffic and may be redundant. Finally, in endpoint-managed flow control schemes each endpoint is responsible for dynamically managing its own flow control credits. This requirement adds to the work that individual endpoints must perform in order to communicate. As the number of nodes increases in the system, this management becomes a substantial effort that requires larger memory and compute resources. These resources may exceed the capabilities of some peripheral devices, thereby preventing their use in the cluster.

### 3.3.2 Per-hop Flow Control

An alternative approach to endpoint managed flow control is for the communication library to perform buffer management on a per-hop basis. In this approach a message is transmitted to the next stage in the communication path as soon as buffer space is available to receive the message. As illustrated in Figure 3.5, the communication library moves data in three phases: sending-endpoint to sending-NI, sending-NI to receiving-NI, and receiving-NI to receiving-endpoint. Each of these phases employs flow-control mechanisms to guarantee that data is transferred reliably from one stage to the next. This approach is commonly referred to as NI-based flow control because the most challenging aspect of the implementation is the transfer of data between NI pairs.

For resource-rich cluster computers the primary advantage of per-hop flow control is that it can greatly simplify the software for communication endpoints. In this scheme an endpoint simply injects a message into its local NI as soon as buffer space becomes available in the NI. From the endpoint's perspective the communication process completes after the injection because the individual network elements in the communication path are guaranteed to reliably transport the message to its destination endpoint. Unlike endpoint-managed flow-control schemes, the per-hop approach does not require endpoints to maintain state information for in-flight messages. This property simplifies the amount of work an endpoint must perform to communicate in a reliable fashion, and is particularly valuable in case where peripheral devices with limited capabilities are being used as endpoints.

Another benefit of a per-hop flow control scheme is that buffer space can be managed dynamically. In this approach a communication element such as a NI makes a decision to accept or reject an incoming message based on whether the element currently has enough buffer space to house the message. Therefore the hardware devices that propagate a message allocate buffer space on demand as needed by applications. An example of how this

trait can be beneficial can be found in a scenario where two endpoints are communicating exclusively with each other at a particular point in time and are not receiving data from other endpoints in the cluster. In this situation the NIs of the elements effectively allocate all of their buffer space for the communication between the two nodes. This buffer space allows more messages to be in-flight between the endpoints at the same time, which improves overlap in the communication pipeline. Endpoint-managed flow control schemes do not allow such dynamic use of resources because allocations are managed at a high level with coarse granularity.

### 3.3.3 Optimistic NI-NI Flow Control

NI-based flow control mechanisms can be implemented in a variety of manners. A popular approach is to employ a credit-based scheme where each NI has a limited number of credits for communicating with other NIs in the cluster. As observed in the endpoint-managed flow control case, this approach may result in a NI unnecessarily delaying a transmission because acknowledgements have not propagated back to the sender. Another approach is to utilize a scheme where the sending NI requests a reservation of buffer space from the receiving NI before a message is transmitted. This approach is useful in times of high network loads because data messages are only transmitted when they can be received. However, this approach has poor performance for the common case where the network is not saturated because a reservation must be acquired before a data message can be transmitted.

An alternative approach to NI-based flow control is to utilize an optimistic transmission scheme. In an optimistic approach the sending NI transmits a message with the expectation that the receiving NI will be capable of accepting the message when it arrives. The receiving NI transmits a positive or negative acknowledgement to the sender depending on whether the message could be accepted or not. If the sending NI receives a positive acknowledgement the buffer space allocated for housing the in-flight message is deallocated. If a negative acknowledgement is received the sender performs a rollback on the outgoing message queue and retransmits the message and all of the following messages that are to the same destination.

An optimistic NI-based flow-control protocol has several benefits. First, similar to a credit-based scheme, an optimistic protocol allows a newly detected message to be transmitted without delay. Second, the optimistic approach does not require any form of credit management. Instead messages must be identified and tracked by the NIs. However, this work is normally required by any NI-based flow control scheme. Third, the NIs naturally allocate buffer space in this approach to meet runtime needs. This trait takes place automatically without explicit signaling between NIs. Finally, the optimistic approach allows the network's delivery latency to be overlapped with useful work. The sending NI can begin transmitting a message at a time when the receiving NI cannot accept it. By the time the message arrives at the receiver it is possible that the receiver will be able to accept the message, thereby reducing the latency of delivery.

## *3.4 Logical Channels*

An important characteristic of resource-rich clusters is that there are multiple communication endpoints in a host that need to interact with the SAN. Since a host generally has more endpoints than NI cards, it is necessary to construct mechanisms that allow the endpoints to share the NI. In traditional approaches, this sharing is performed in the kernel

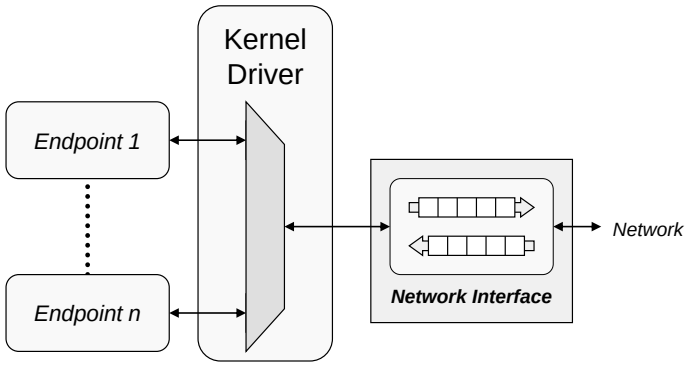


**Figure 3.6:** The traditional approach to providing shared access to a network device through the use of the kernel.

by constructing multiple virtual network interfaces for end applications. Unfortunately this approach is inefficient for resource-rich clusters because it is difficult to present these virtual interfaces to peripheral device endpoints in an efficient manner.

Without kernel-based NI management, it is necessary to implement synchronization mechanisms in the individual endpoints to guarantee that the NI is accessed in a mutually exclusive manner. Utilizing explicit signaling between endpoints is complex and impedes performance. Therefore we propose moving the task of managing shared access to the network into the NI. In this approach the NI employs multiple message queues that are referred to as logical channels. Each endpoint has exclusive ownership of a small number of the NI's logical channels. The endpoint utilizes these logical channels as virtual interfaces for communication with the network. The task of mapping the logical channels onto the physical network is dynamically performed by the NI. In addition to providing a sharable means of low-latency communication, logical channels can also be utilized by applications to provide isolation between different types of network data streams and allocate bandwidth among peripheral devices.

### 3.4.1 Sharing Network Access through Kernel Management

For traditional networks such as Ethernet, the kernel is utilized as a means of sharing a physical NI card with multiple applications. As depicted in Figure 3.6 the kernel has exclusive ownership of the NI and provides virtual communication interfaces for multiple application endpoints. The kernel therefore must merge the messages injected by endpoints into a single outbound NI queue and distribute incoming messages from the network to the proper endpoints. In addition to providing a scalable means of sharing the NI, this approach protects end applications from each other by insulating the applications from the low-level hardware.

Utilizing the kernel as a means of sharing the NI is impractical for resource-rich clusters due to the types of network interactions that are utilized in these clusters. The primary problem with relying on the kernel to manage the NI is that the communication interfaces provided by the kernel are designed to operate with host-level endpoints, not peripheral device endpoints. Adapting a peripheral device to operate with these interfaces is difficult and inefficient. The peripheral device would have to route all of its network transactions through the kernel and utilize costly interrupts to invoke the necessary kernel operations. This process requires extra data copies and taxes the memory and I/O systems of the host.

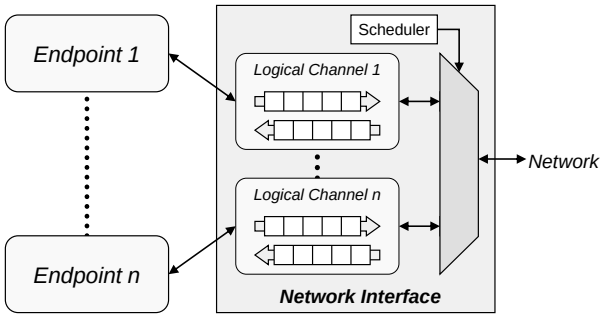


**Figure 3.7:** Utilizing multiple logical channels in the NI to provide shared access to the network.

Another disadvantage of utilizing the kernel to manage the NI is that host-level endpoints must invoke kernel calls for network operations. Since kernel calls can be relatively expensive operations, it is beneficial if shared access to the NI can be accomplished without involving the kernel driver.

### 3.4.2 Sharing Network Access through Logical Channels

Another means of sharing the NI with multiple endpoints is simply to remove the dependencies that exist between endpoints that interact with the NI. One such approach is to implement a small number of independent message queues or logical channels in the NI. Each of these logical channels is assigned to an endpoint in the host when the system is initialized. Because an endpoint has exclusive ownership of its logical channel(s), it can send and receive message without having to synchronize with other endpoints in the system. The NI in this approach is responsible for mapping the logical channels onto the physical network at runtime through the use of a simple scheduling algorithm. An example of this approach is illustrated in Figure 3.7.

There are several benefits to using logical channels as a means of providing shared access to the NI. First, this approach removes the need for any form of direct synchronization between endpoints that are interacting with the NI. An endpoint can begin injecting data into the NI as soon as buffer space is available in its logical channel. Second, endpoints interact directly with the NI. Unlike kernel-managed approaches, an endpoint transfers data directly into the NI without intermediate buffering. Finally, this approach provides a simple interface for communication that can be implemented for many peripheral devices without complex management mechanisms.

There are two primary disadvantages to utilizing logical channels in the NI. First, there is a finite amount of buffer space available in the NI for implementing logical channels. As the number of logical channels in the NI increases, the buffer capacity of each logical channel decreases. Therefore it is expected that most resource-rich cluster users will allocate only a few logical channels in the NI (i.e., roughly one per endpoint). Second, the presence of multiple logical channels in the NI has a negative impact on the performance of the NI. Because the NI must spend time managing each logical channel, the NI's workload increases as more logical channels are added to the NI. Additionally, NI firmware becomes more complex when it is switched from servicing a single queue to multiple queues. This complexity results in extra NI operations which detract from performance.

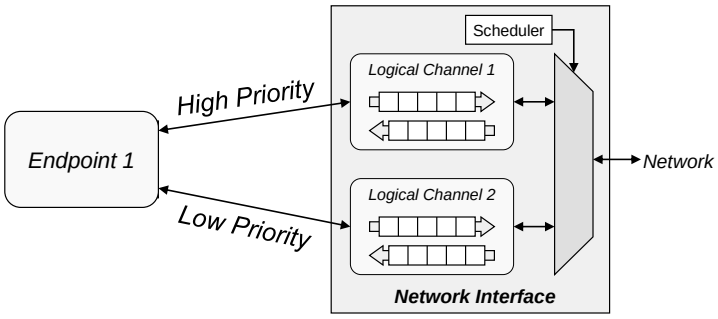


**Figure 3.8:** Utilizing multiple logical channels to prioritize messages.

### 3.4.3 Application Level Use of Logical Channels

In addition to allowing multiple endpoints to share the same NI, logical channels can be utilized by end applications as a simple means of separating traffic streams. For this use an application requests two or more logical channels from the NI and assigns different traffic streams to each channel. Data streams on different logical channels are isolated from each other due to two properties of logical channels. First, each logical channel has a private allocation of buffer space in the NI. Therefore if one traffic stream saturates its logical channel's buffer space, other logical channels are not affected. Second, in-order delivery in the communication library is guaranteed only for messages that belong to the same logical channel. This property is necessary in order to allow the NI to implement a fair scheduling algorithm where each logical channel has equal access to the NI. The result is that a message injected into an empty logical channel does not have to be delayed until all of the messages in other queues are transmitted.

An example of how the presence of multiple logical channels in the NI can be exploited by an end application is illustrated in Figure 3.8. In this example an endpoint obtains two separate NI logical channels for two types of network traffic. The network bandwidth made available for each logical channel is controlled through a scheduler implemented in the NI.

## *3.5 Active Message Programming Interface*

One of the defining characteristics of a communication library is the programming interface that is provided to the end user. Users of resource-rich clusters require a flexible programming interface that can easily be extended to support new functionality. As a means of addressing this need we propose constructing the communication library with two types of programming interfaces: one that employs active message style processing (described in this section) and another that provides a means of interacting with remote memory (described in the following section). For the active message interface each communication endpoint is equipped with various function handlers for processing incoming messages. Whenever an endpoint injects a message it specifies the function handler the receiving endpoint should use to process the message when it arrives. In addition to providing a powerful means of controlling computations in a distributed processing environment, the active message programming interface is well suited to controlling peripheral devices in a resource-rich cluster. In this effort peripheral device functionality is encapsulated as a set of active message function handlers that all endpoints in the cluster can utilize.

### 3.5.1 Active Message Operation

The fundamental concept of active messages is that a message contains both application data and information describing how the receiver should process the message. While active network research [88] has discussed encoding complex processing instructions into active messages, a more common approach is for endpoints to be equipped with predefined methods for processing messages. These methods are commonly referred to as function handlers. When the communication library is initialized each endpoint publishes a list of its function handlers to other endpoints in the system. At runtime when an endpoint injects a message into the communication library it must specify the function handler the receiver should use to process the message. Endpoints are responsible for monitoring incoming message queues and processing new messages with the appropriate function handler.

The appeal of an active message interface is that it provides basic programming mechanisms that are both powerful and flexible. As opposed to simply transferring data between endpoints active messages provide a means of invoking actions at remote endpoints. These actions can be utilized in an active manner to remotely control the behavior of an endpoint. For example a message handler can be designed to spawn, modify, or terminate a computation in an endpoint. With these types of operations a user can directly control the flow of computations in a distributed system. Active messages can also be utilized in a passive manner where a remote endpoint's state is not affected by the execution of a handler. For example, a handler can be designed to simply return the remote endpoint's current dataset to the sender of the message. From the remote endpoint's perspective the processing of the function handler takes place in the background and does not affect the endpoint's main thread of execution.

The original active message specification [27] is not directly applicable for resource-rich clusters because it is only designed to operate with homogeneous endpoints. Therefore it is necessary to construct a more robust specification that allows diverse endpoints to interact with the active message interface. Three issues must be addressed in this specification. First, function handlers must be managed in a dynamic fashion by the communication library. It is not practical to statically configure endpoints with a list of the cluster's handlers because endpoint software would have to be recompiled every time a new application defined new handlers. Second, handlers should be referenced with useful labels, such as string and integer identifiers. In addition to being portable, these identifiers help make programs more readable (e.g., referencing a handler by the string "handler_compute_PI" has more meaning than a pointer to the handler's virtual memory address). Finally, active messages should be formatted in a manner that is interpretable by endpoints with different byte orders and word alignments. Constructing a single message format that takes into account these characteristics provides standardization among endpoints and allows an endpoint to transmit a message without having to know the processing characteristics of the destination endpoint.

### 3.5.2 Utilizing Active Messages with Peripheral Devices

The active message programming abstraction is particularly useful for resource-rich clusters because active messages can be used as a simple but powerful means of controlling peripheral devices. In this approach a set of active message function handlers are defined for all of the actions that a peripheral device can perform. Endpoints in the cluster can therefore trigger an operation at a peripheral device by transmitting a properly formatted active message to

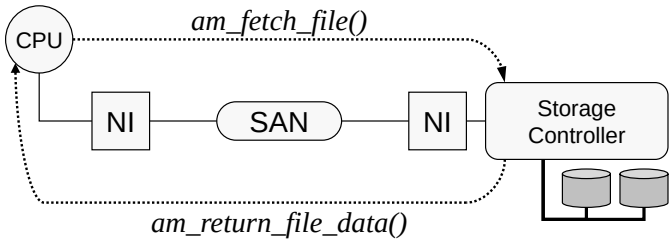


**Figure 3.9:** Active messages can be used to facilitate an API for a peripheral device.

the device containing a reference to the function handler that needs to be invoked. Figure 3.9 illustrates an example of how a host-level endpoint can interact with an intelligent storage controller at a remote host using the active message programming interface. In order to obtain data from a desired file, the host CPU transmits an active message that contains the name of the file and the function handler id `am_fetch_file()`. Upon receiving this message the storage controller accesses the file and generates an active message with the handler `am_return_file_data()` and the requested data. The transaction completes when the host CPU endpoint receives this reply and stores the data accordingly.

Using an active message programming interface to control a cluster's peripheral devices is beneficial for a number of reasons. First, it is relatively easy to integrate new peripheral devices into the cluster using this interface. Designers simply construct a series of card-specific active message function handlers for a peripheral device and provide references for the handlers to application designers. Second, the active message interface serves as a universal communication substrate upon which multiple APIs can be layered. In this system each peripheral device has its own API that is comprised of card-specific active message handlers. Endpoints therefore invoke a peripheral device's API operations by transmitting the corresponding active messages using the communication library's message passing functions. Finally, the active message interface is beneficial for controlling peripheral devices because it allows an endpoint to utilize a peripheral device no matter where the resources are physically located. Since API operations are separated from communication mechanisms, users can issue API operations knowing that the communication library will automatically perform any routing in the cluster that is necessary.

## *3.6 Remote Memory Programming Interface*

The second programming interface proposed for a resource-rich cluster's communication library is one that allows an endpoint to directly interact with the memory of a remote host. This remote memory interface is designed to provide an efficient means of transferring data from one endpoint to another. A remote memory programming interface can also be utilized as a means of performing custom interactions with a cluster's peripheral devices. This functionality is especially beneficial because it can be used to allow an endpoint to control a peripheral device for which it is impossible to construct endpoint software. Issues involved in implementing a remote memory interface include integrating the interface into a library that also supports active messages, and providing the functionality to translate an endpoint's virtual address space into a physical address space.

### 3.6.1 The Need for a Remote Memory Interface

While active messages provide a flexible communication interface for end users, there are certain operations for which active messages are not ideal. For example, consider the case where an application needs to transfer a large block of data from one endpoint to another. In the active message approach the data is encapsulated in an active message that is marked with a data transfer function handler. The receiver processes this message by copying the message's payload data to the memory location specified in the message's arguments. This process is inefficient because two transfers are involved in the receiving endpoint: one from the NI to the endpoint's incoming message queue and another from the message queue to the target address. With a remote memory programming interface it is possible for the NI to transfer the data directly to the message's target memory address.

Remote memory operations are also valuable in resource-rich clusters because they can be used to support low-level interactions with remote peripheral devices. The architecture of several peripheral devices makes it impossible to construct endpoint software that would allow these devices to participate as intelligent resources in the cluster. For example, video display adaptors are generally designed as data sinks and therefore it is unlikely that endpoint software can be constructed for such adaptors. However it is still desirable for other resources in the cluster to be able to interact with the adaptor. With a remote memory interface it is possible for an endpoint to transmit image data into the video adaptor's frame buffer. These forms of direct memory transactions can be useful in a number of resource-rich cluster applications where data must be deposited into distributed memory locations in an efficient manner.

### 3.6.2 Remote Memory Interface

From an end user's perspective a remote memory interface is relatively straightforward. The user supplies the interface with the source and destination addresses, the direction and size of the transfer, and the identifier of the remote endpoint. The communication library is then responsible for transferring the block of data utilizing the most efficient means available. In the case of multiple transactions, remote memory transfers are completed in the order that they are issued. Remote memory interfaces generally allow both read and write operations. Write operations are simpler to implement as data is simply streamed from the sender's address space to the receiver's. Read operations are more complex as the sender must issue a message that fetches data from the receiver's address space. Results are returned in a reply message and written into the sender's address space.

A host system operates with two related address spaces: virtual memory addresses and physical (or bus) memory addresses. The API for a remote memory interface must be designed to allow users to universally reference memory distributed throughout the cluster. As a means of simplifying the interface for end users utilizing virtual memory references are preferred since a memory reference is the same in both the host where the memory resides and remote endpoints. As a consequence it is necessary for the communication library to be capable of internally transforming virtual addresses to physical addresses that the NI can utilize. The library must also provide mechanisms to prevent a memory region from being moved by the kernel (e.g., a page fault) during a memory operation. Finally, it is beneficial for a remote memory interface to be able to operate with physical addresses, in order to provide efficient direct access to memory-mapped devices.

## *3.7 Related Work*

Various aspects of this thesis are related to themes found in other research projects. A common goal of all these efforts is to enhance cluster computer performance by incorporating powerful peripheral devices within the hosts. Researchers have designed custom I/O architectures to support this functionality as well as specialized software to integrate specific peripheral devices into the communication model. This thesis is distinguished from past work in that it provides a general framework for integrating all manner of peripherals into a low-latency message layer. Device-specific functionality is separated from network-specific functionality to produce an extensible design and significantly improve the productivity of the application designer with minimal sacrifices in performance. The following efforts represent state-of-the-art research being performed involving resource-rich cluster computers.

### 3.7.1 InfiniBand

Industry is currently developing a new generation of I/O fabric called InfiniBand (IB) [44] that can potentially serve as a means of constructing resource-rich cluster computers. IB is primarily designed as a turnkey solution for a number of high-end server issues. It provides a high-performance communication substrate that functions as a system area network, a storage area network, and a distributed I/O system. In addition to featuring expandable multi-gigabit links, IB defines a protocol for efficient communication between peripheral devices and host CPUs. This protocol could therefore be utilized by end users to allow peripheral devices to be integrated into the cluster's computational model. Therefore IB represents a promising communication substrate for resource-rich cluster computers in the near future.

A fundamental difference between the work presented in this thesis and InfiniBand can be found in the hardware architectures used for these systems. In the work presented in this thesis, it is assumed that cluster computers will be constructed with commodity hardware that is currently available. This approach utilizes existing hardware and defines flexible mechanisms for addressing the performance obstacles of the hardware. In contrast, IB is a complete overhaul of the I/O architecture found in current generation clusters. With the freedom to redesign the low-level architecture of the cluster computer, IB designers constructed a new hardware environment that is conducive to high-performance communication. The difficulty in this approach is public acceptance: the success of IB as a communication substrate depends on the generation of new hardware products that provide better performance than current products. In comparison, the work in this thesis utilizes current generation hardware and can be adapted to exploit gains in faster network substrates as they become available.

### 3.7.2 Extensions to the GM Message Layer

In recent years Myricom's GM message layer has become the de facto standard for traditional clusters interconnected with Myrinet hardware. GM exhibits a number of basic characteristics that make it a desirable starting point for constructing a message layer for resource-rich clusters. In addition to utilizing NI-based flow-control mechanisms, GM supports multiple concurrent users of the NI through the use of multiple work queues. While GM does not specifically support active messages, it provides a generic programming interface that allows other APIs to be layered on top of it. GM also provides mechanisms for

low-level interactions with the virtual memory system, which can be extended to provide remote memory operations.

While GM can be extended we note that there are fundamental design issues that make the adaptation of the message layer to resource-rich clusters non-trivial. The primary difficulty is that the basic means for a communication endpoint to interact with the message layer is through a work queue. In this approach an endpoint inserts a reference to a message that needs to be injected. When the NI is ready it processes the work entry by pulling the message into the NI. It then inserts a notification message into the endpoint's completion queue that specifies that the host memory housing the message can be reused by the application. While suitable for host-level endpoints, this process may not be appropriate for some peripheral devices because it requires the peripheral device to maintain a block of data until the NI has retrieved it. Peripheral devices generally have limited memory and resources to manage such interactions.

### 3.7.3 OPIUM

GM has been extended in previous work to allow the NI to directly interact with multiple peripheral devices. In the OPIUM [20] project researchers examined the extension of GM to support SAN interactions with a specific SCSI card. The goal of this work is to minimize the number of traversals that take place across the PCI bus for servicing network requests for file data. The researchers accomplished this task by modifying the storage card's device driver so that it could issue DMA operations to route file data directly to a buffer located in NI card memory. In later work opium was modified to allow the NI to directly write data into a video display card's frame buffer [38].

While Opium provides the first steps in allowing peripheral device interactions with the SAN, the work is directed at providing an ad hoc solution for two specific devices. While the modifications that allow the host to control SCSI interactions with the network is certainly useful for network attached storage efforts, the work is card-specific and may not be suitable for other peripheral devices that could be used in the cluster. Likewise, the work with integrating a video display card into the communication library does not demonstrate an interaction with an intelligent peripheral device, because a display card's frame buffer can trivially be written by any PCI device in a host. This work however does provide a motivation to improve the flexibility of the communication library in order to allow peripheral devices to be utilized in an efficient manner by cluster applications.

### 3.7.4 Adaptive Computing Machines

Another area of work that is related to this thesis is the field of Adaptive Computing Machines (ACMs). In ACMs a number of field-programmable gate arrays (FPGAs) are utilized as a means of processing an application with dedicated hardware [30]. In this approach the FPGAs are configured to emulate application-specific circuitry that can rapidly perform an application's computations. Unlike ASICs which cannot be reprogrammed, FPGAs can easily be configured to emulate different circuits as needed by the application. While ACMs are not particularly useful for general-purpose applications, they can be valuable for applications that require complex computations be performed in real time [92, 57].

Initial work in ACMs resulted in custom hardware that employed arrays of FPGAs [39]. Observing that these systems were expensive to construct, researchers in the late 1990's began investigating the use of multiple commercial FPGA cards to function as an

ACM. In the Tower of Power project [52], sixteen x86 workstations were equipped with commercial FPGA cards and linked using a Myrinet SAN. The researchers investigated the use of existing Myrinet software to allow data to be transmitted between FPGA cards [11]. This effort resulted in the computational environment where researchers could effectively utilize the distributed FPGA cards as part of an ACM.

One of the hardships that researchers had to face in the Tower of Power project is transporting data between FPGAs in the cluster. Rather than implement new communication software the researchers layered their programming interface on top of a Myrinet implementation of MPI. The researcher's software therefore utilizes the host CPU to manage application interactions with an FPGA card. While simplifying the design effort, this approach delays communication and results in extra traversals of the host's I/O bus. Additionally, selecting MPI as the base programming interface makes it challenging to modify the system to support direct interactions between the FPGA card and the NI. MPI endpoint software is complex and therefore nontrivial to implement for an FPGA card. However, this work indicates that there is a definite interest in utilizing peripheral devices in a cluster to perform custom computations.