CHAPTER IV

MESSAGE LAYER IMPLEMENTATION: GRIM

The design considerations outlined in the previous chapter have driven the implementation of an extensible message layer for resource-rich clusters. This message layer is known as GRIM: the General-purpose Reliable In-order Message layer. GRIM has evolved considerably since its initial development in 1997, but has always provided three basic features: NI-directed flow-control, NI-based logical channels, and an active message style programming interface for interactions between cluster resources. This work was extended in later versions to provide an additional remote memory interface for efficiently transferring data between endpoints. GRIM has been used to incorporate multiple peripheral devices into the cluster computing environment. This chapter examines the core functionality of the GRIM library, specifically focusing on low-level implementation details that shaped the library. The core's end-to-end performance for host-CPU interactions is presented in the next chapter, which is followed by details of the library's use in peripheral device interactions.

4.1 Overview of GRIM

GRIM is a message layer for resource-rich cluster computers that in the current version utilizes a Myrint SAN for interconnecting host systems. The GRIM communication library is comprised of user-space software, kernel-space device drivers, and peripheral device firmware. The library currently utilizes the Linux 2.4 kernel, although previous kernels have been used during GRIM's evolution. In order to minimize the impact of an ever-changing Linux kernel, the majority of GRIM's functionality is constructed in user-space software and NI-based firmware. In recent versions of GRIM, the LANai 4 NI firmware has been ported for use with the newer LANai 9 version of the Myrinet NI card. This adaptation has resulted in significant performance improvements due to advances in the LANai's architecture.

The organization of GRIM's core components is depicted in Figure 4.1. Starting at the lowest level of the software, GRIM utilizes a NI-based reliable transmission protocol for delivering messages in order between NIs. This protocol is optimistic in that messages are transmitted with the expectation that the receiver can accept the message. Messages are supplied to the reliable transmission protocol from a small collection of logical channels located in NI memory. Each logical channel provides a virtual communication interface for an endpoint in the host and serves as a place for buffering messages that are in transit. At the application level an endpoint can utilize two programming interfaces for interacting with a logical channel. The active message interface allows the sender to mark an outgoing message with the function handler the receiver should use to process the message. This interface provides a queue in the endpoint for buffering incoming message that cannot be processed immediately by the endpoint. The second programming interface is for remote memory operations. Memory used with this interface must be registered with the communication library. This interface provides multiple mechanisms for translating virtual addresses to physical addresses, and allows the NI to process incoming remote memory messages directly.

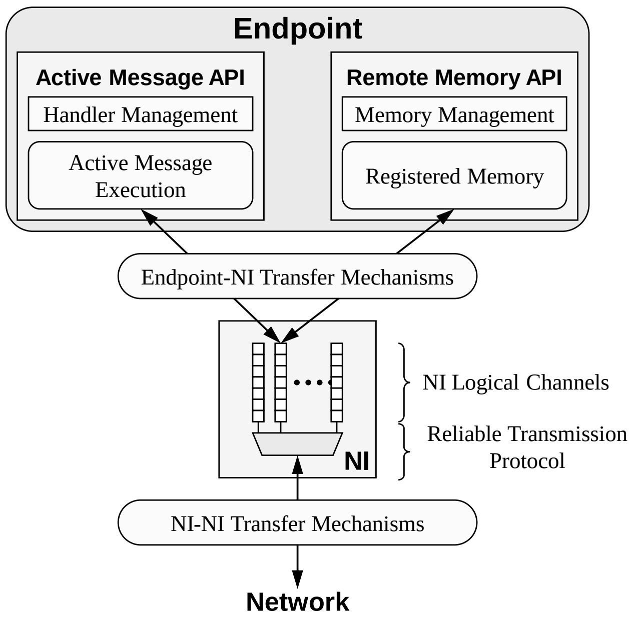

Figure 4.1: GRIM utilizes logical channels and a reliable transmission protocol at the NI and provides two different programming interfaces for end applications.

4.1.1 Message Structure

The fundamental unit of communication in the GRIM communication library is a data message. GRIM uses a single message format for all of its operations and therefore it is instructive to examine the common message format. A data message in GRIM comprised of two portions: a message header and payload data. The message header consists of eight 32-bit words that are formatted in network byte order (i.e., big endian). The first four words contain information necessary for delivering the message. These words are arranged in a manner that allows a receiver NI to begin processing an incoming message as soon as possible with consequent reductions in overall latency. The remaining four words allow users to supply up to four user-defined 32-bit data values in each message. While the inclusion of these fields in every message header increases GRIM's minimum message size, the fields are frequently used in a variety of manners and simplify the interface for the end user. Following the message's header is payload data. This region is defined entirely by the user.

Figure 4.2 provides the format of a GRIM data message. The individual fields in a message are defined as follows.

- **GRIM ID**: This field identifies a message as belonging to the GRIM communication library. A NI in GRIM only examines messages that are labeled with this identifier. GRIM is registered with Myricom and has been assigned the identifier 0x0636.
- **NI Token**: The token ID is supplied by the NI and utilized to reference an in-transit message.
- **NI Sequence Number**: The sequence number is created by the sending NI and utilized by the receiving NI to verify delivery order in the reliable transmission protocol.

32b			16b	1b
GRIM ID			NI Token	NI Sequence
Message Type (6)	Payload Correction (2)	Multicast Tree (8)	Source Endpoint	
Payload Word Length			Destination Endpoint	
Logical Channel			Active Message Function Handler	
User Argument[0]				
User Argument[1]				
User Argument[2]				
User Argument[3]				
Message Payload(0 - 64,572 Bytes)				

Figure 4.2: GRIM uses a single message format for all transactions in the communication library.

- **Message Type**: The type field is used to distinguish between different forms of messages used in the library. Message types include active messages, remote memory operations, NI control messages (e.g., ACK or NACK), and multicast operations.
- **Payload Correction**: Internally GRIM aligns payload data on 32-bit boundaries. The payload correction value is used to truncate the length of the payload to match the number of bytes specified by the sending endpoint.
- **Multicast Tree**: This value is used in multicast operations to identify the multicast tree to which a message belongs. Multicast operations are discussed in Section 8.1.
- **Source Endpoint**: The source field identifies the endpoint that created the message.
- **Payload Word Length**: This field specifies the number of 32-bit words that are in the payload section of the message.
- **Destination Endpoint**: This value identifies which endpoint in the cluster to which the message should be delivered.
- **Logical Channel**: An endpoint that transmits a message can assign a 16-bit logical channel identifier to the message. This identifier is used to group messages that must be delivered in order by the message layer.
- **Active Message Function Handler**: This value identifies which active message function handler should be used to process the message at the receiving endpoint.
- **User Arguments[0-3]**: Users can specify up to four 32-bit arguments to be included in an active message. These fields are utilized in remote memory operations to specify the addresses to be utilized in a data transfer.

Internally GRIM uses an abbreviated version of the message header for control messages (ACKs and NACKs). Control messages are only 8 bytes long and contain basic information to allow updates in the reliable transmission protocol.

4.2 NI-Based Reliable Transmission Protocol

One of the key characteristics of GRIM is the use of a reliable transmission protocol for transferring messages between NIs in the cluster. GRIM utilizes a variation of the "go-back-n" protocol that optimistically transmits a message with the expectation that the receiver will be able to accept the message when it arrives. If the receiver cannot accept the message, the protocol automatically performs rollback on the sender's message queue and retransmits messages as needed. In order to facilitate this operation GRIM utilizes control messages and maintains state information for each message queue. Since control messages utilize the same network as data messages, it is possible for a poorly designed system to reach deadlock. GRIM avoids this condition by buffering outgoing control messages when a response to an incoming message cannot be transmitted due to a busy outgoing link. Performance measurements of GRIM suggest that the NI-based reliable delivery protocol is advantageous over other approaches. When compared to system employing host-based flow control, GRIM allows endpoints to inject a greater number of outstanding messages, thereby reducing the sending endpoint's injection overhead.

4.2.1 Protocol

The NI-based reliable transmission protocol implemented in GRIM is a variant of the traditional "go-back-N" protocol [87] for retransmissions and operates as follows. The successful transmission of a message is depicted in Figure 4.3(a) with three steps. (1) When a sending NI observes a new message to send it marks the message with the next sequence id for the destination NI and a token id that can be used to reference the message. It then transmits the message to the destination and increases the sequence number register for the destination. (2) When the message reaches its destination NI, the receiver NI verifies that the message's sequence number matches the expected value for the sender. If it does and the receiver has enough buffer space, the message is accepted and a positive acknowledgement (ACK) with the data message's token id is transmitted to the sender. (3) When the ACK reaches the sender NI, the NI uses the token id to mark the corresponding message in the message queue as acknowledged. If the message is the oldest outstanding message, the NI walks through the queue structure freeing buffer space for all acknowledged messages until it reaches an unacknowledged message, a message that has not been transmitted, or the back of the outgoing queue. With this protocol the sender is allowed to have multiple messages to a destination in-flight at the same time as illustrated in Figure 4.3(b).

In the case where the receiver cannot accept an incoming message due to a lack of buffer space, the protocol uses a negative acknowledgement (NACK) control message to force the sender to retransmit messages. Figure 4.3(c) illustrates such a case where a sender optimistically transmits three messages with sequence numbers and token ids of 0, 1, and 2. After the first message is accepted the receiver is unable to accept data due to a lack of buffer space and must transmit a NACK for message 1. The receiver at this point drops incoming messages from the sender until message 1 is received and buffer space is available. When the sender receives a NACK it must rollback the outgoing queue to the message referenced in the NACK. It then retransmits the message and all following messages in the outbound queue that are for the same destination. Messages for other destinations are not retransmitted. The procedure is repeated until all messages are reliably delivered.

The implementation of this protocol in GRIM takes advantage of Myrinet's reliability guarantees and the fact that source routed messages do not get reordered in the network.

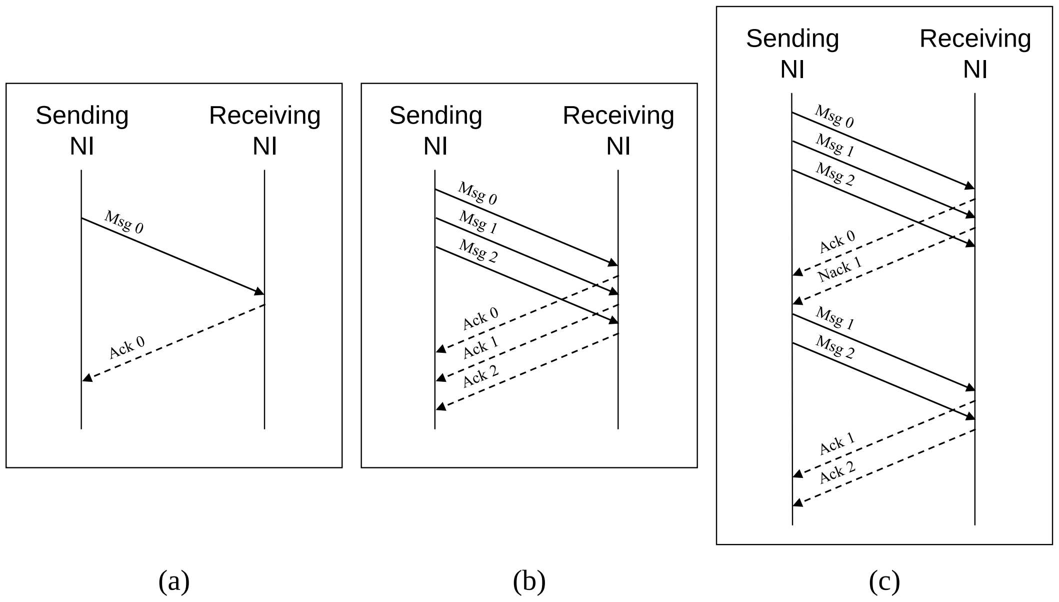

Figure 4.3: (a) Acknowledged transmission of a single message between NI pairs. (b) The optimistic transmission and acknowledgement of three messages. (c) The optimistic transmission of three messages with retransmission due to the lack of buffer space.

While other network substrates do not exhibit these characteristics, the implementation can be extended to function under different operating conditions without significant changes to the protocol. For networks that can re-order messages (such as an Ethernet LAN with multiple routers), the sequencing portion of the protocol preserves in-order delivery. The arbitrary dropping of packets on the other hand requires the protocol to be modified with timeout mechanisms. These mechanisms automatically retransmit data and control messages after a specified amount of time under the assumption that the network has lost a message.

4.2.2 Managing In-flight Messages for Different Queue Mechanisms

An important part of implementing a NI-level reliable transmission scheme is constructing mechanisms that allow the NI to manage multiple in-flight messages. These mechanisms require the sending NI to maintain a database of in-flight messages that is populated with information that can be used to coherently perform rollback on a message queue when a message needs to be retransmitted. This work is highly dependent on the types of queuing mechanisms that are used to buffer messages in the sending NI. Over its evolution, three different styles of queuing have been used for GRIM, as illustrated in Figure 4.4(a-c). In a slotted approach (a), queue buffer space is evenly divided into fixed-sized slots for housing individual messages. An append-style approach (b) differs in that messages can be placed in the queue without any gap between successive messages. Finally, in a hybrid-approach (c), a combination of the previous two mechanisms is used. In this approach a message's header is stored in a slotted message queue while its payload is stored in a separate append-style queue.

The advantage of a slotted approach to queuing messages is that messages always begin at specified locations in the queue. In addition to simplifying the management of the

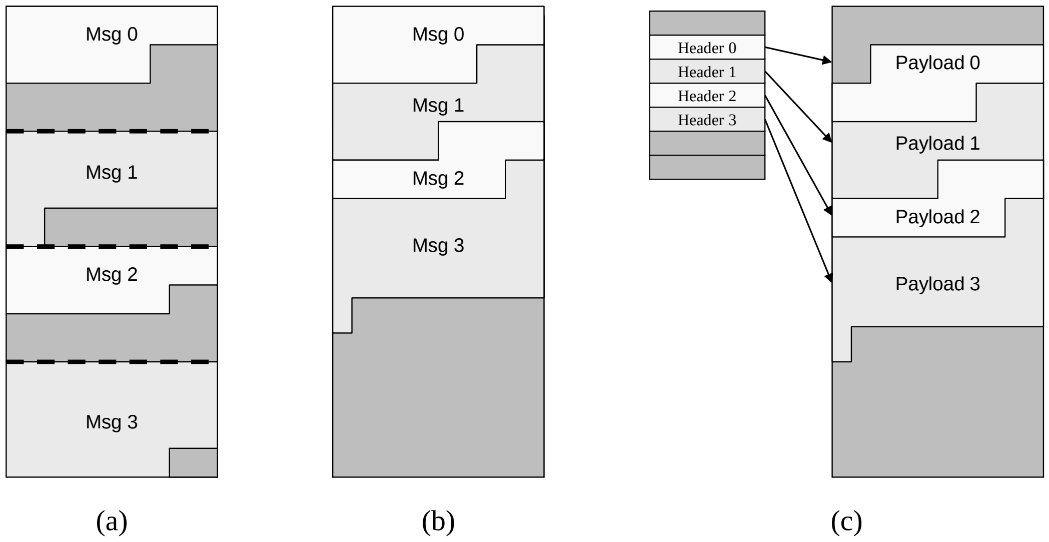

Figure 4.4: Three approaches to queue buffer management include (a) a fixed-sized slot queue buffer, (b) an append-style approach, and (c) a hybrid approach.

queue, the NI can easily store a message's state information in its message queue slot. For example, the first word in a message slot can be reserved for housing the message's current acknowledgement status (e.g., not sent, sent, acknowledged, or unused). The sending NI can then easily walk through the queue structure when rollback is performed and make necessary state updates by modifying specified values in each slot. The downside of this approach is that queue buffer space is not used efficiently when messages are not the maximum transfer size. Because of the limited amount of NI buffer space, this approach is not utilized in GRIM.

The append-style message queue makes more efficient use of a queue's buffer space by allowing messages to be placed in the queue without wasting space between messages. The difficulty with this approach is that messages are not stored at fixed locations in the queue. Therefore accessing a particular message in the queue is non-trivial because the NI must sequentially walk through the queue, examining each message to determine the starting address of the next message. This operation is expensive and makes storing state information in the message queue impractical. Instead, state information for in-transit messages can be stored in a separate data structure by the sending NI. Recent versions of GRIM employ an append style of queuing and store message information in a structure called a scoreboard. When a NI detects a new message in the outgoing queue it records information about the message (such as its memory location) in a new scoreboard entry. Because scoreboard entries are at fixed offsets, the NI can easily walk through the scoreboard when processing incoming control messages.

The last style of message queue used in GRIM is a hybrid-approach where a slotted queue is used to store a message's header and an append-style queue stores a message's payload. This approach is advantageous because in-flight messages can easily be managed using information in the slotted header queue while at the same time message payloads are stored efficiently. Although the hybrid approach was used in early versions of GRIM, it had to be abandoned due to a few shortcomings. The first issue is that endpoints in this approach must maintain two sets of queue pointers to interact with an outgoing message

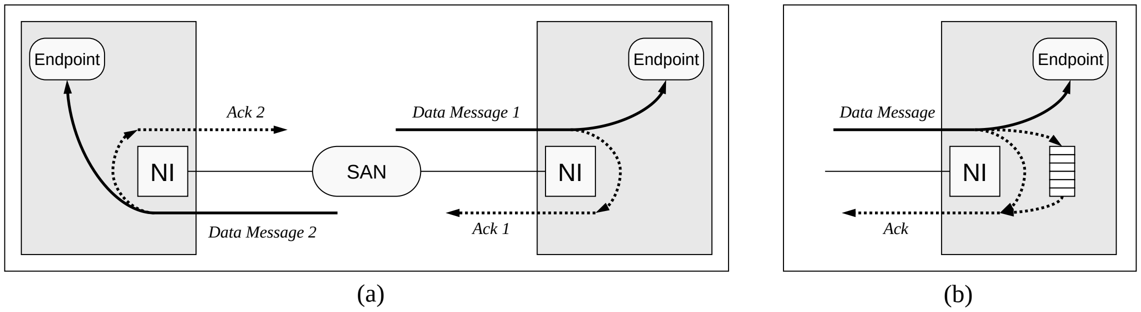

Figure 4.5: The use of control messages can result in deadlock. A cycle is formed in (a) when two nodes transmit data messages to each other at the same time. Deadlock can be prevented by buffering control messages (b) when the outgoing link is not available.

queue. This requirement complicates endpoint software and does not match the design goals of a resource-rich cluster. A second and more serious issue is that it is difficult to protect against deadlock. Since a message is housed in two separate memory regions, the NI must perform two separate DMAs of the message to the network. If the NI firmware blocks the transmission of the second DMA until the first DMA completes, a cyclic dependency is formed and it is possible to reach deadlock. This form of deadlock was observed in the early versions of GRIM and therefore the hybrid approach was dropped in favor of the previous append-style approach. The append-style approach allows a message to be transferred to the network with a single DMA.

4.2.3 Avoiding Deadlock Caused by Control Messages

An important aspect of implementing a communication protocol is constructing it in a manner that does not lead to deadlock. While the network itself may be deadlock free, poorly designed reliable transmission protocols for the NI may result in cyclic dependencies between the NIs that prevent forward progress in the system. The primary hazard is that a NI must inject an ACK or NACK message back into the network upon receiving a data message. If the NI cannot accept a new message until the control message is dispatched, a dependency is formed between incoming and outgoing network links. An example of how this dependency can lead to deadlock is pictured in Figure 4.5(a). In this example two NIs transmit data messages to each other at the same time in a congested network. Both NIs accept their incoming messages and must transmit reply messages for the receive process to complete. However, because one NI cannot proceed until the other completes the injection of the control message, neither can make progress and the result is deadlock. This phenomenon was infrequently observed in early versions of GRIM, even with small network configurations.

One means of deterring this form of deadlock is to provide buffering within the cycle. As illustrated in Figure 4.5(b), GRIM employs a special queue for buffering control messages that cannot be injected into the network due to a busy outgoing link. When an incoming data message is processed by a NI, an ACK or NACK message is inserted into the control message queue if the outgoing link is busy or the control message queue is already populated. The NI's firmware is designed to transmit buffered control messages as soon as the outgoing link becomes available. This method of preventing deadlock relies on the consumption

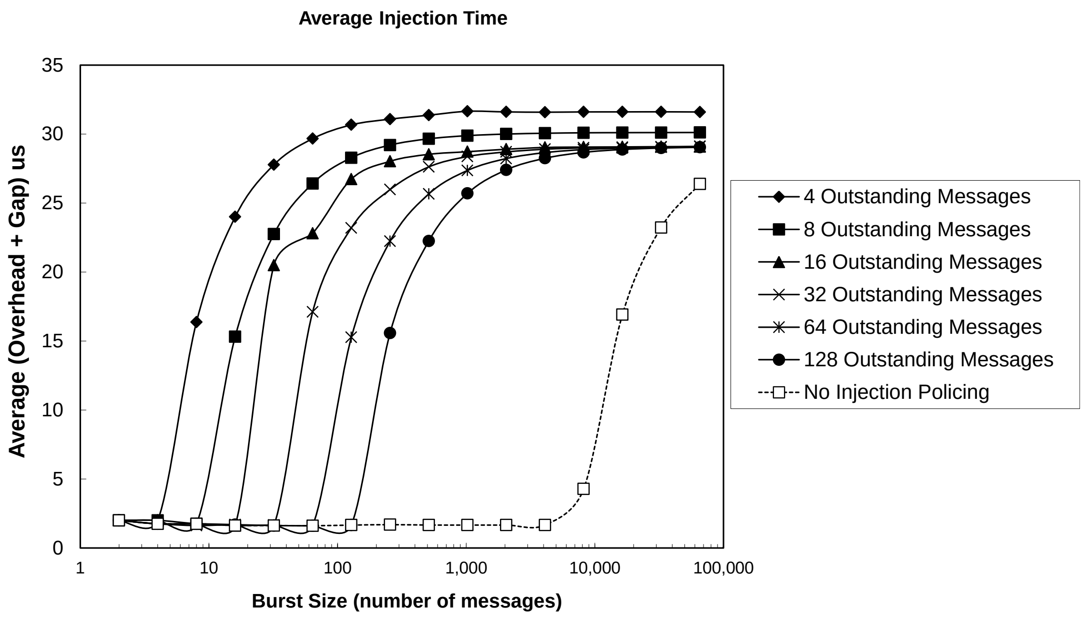

Figure 4.6: Injection policing effects of a credit-based flow control scheme implemented on top of the optimistic NI-based scheme used in GRIM. Performance is measured as the average message injection overhead time for null length messages.

assumption [67] that is basis of most deadlock prevention schemes.

4.2.4 Observed Advantages to NI-based Flow Control

One of the arguments for employing flow control in the NIs is that buffer space can be used dynamically as needed by applications. In this scheme a receiving NI rejects a message only if the intended endpoint lacks buffer space for accepting the message. In comparison, endpoint-based flow-control schemes generally reserve buffer space across the entire communication path before a message can be transmitted. This reservation results in injection policing that can limit performance. As a means of investigating the effects of injection policing, a benchmark program was constructed for GRIM to simulate an endpoint-based flow-control scheme. In this test an endpoint transmits a large number of messages to another endpoint, which in turn transmits all the messages back to the sender as soon as they are received. In order to observe policing effects, the sending NI is limited from having no more than a fixed number of outstanding messages in-flight to the destination at any time. The benchmark measures the amount of time required for the sender to inject a burst of null-length messages. This value is divided by the burst size to determine the average injection overhead for a single message in the burst transfer. The benchmark is run multiple times, varying the burst size and maximum number of outstanding messages the sending endpoint is allowed to have at any time.

The results of this experiment are presented in Figure 4.6 for hosts that allow from four to an unbounded number of outstanding messages. As expected the average message injection overhead for each test remains low until the injection burst size exceeds the number of outstanding messages the sender is allowed to have. After this point injection overhead rapidly increases to a steady-state value. While sharp, this increase is not instantaneous because the receiver injects credit-replenishing replies at the same time the sender injects outgoing messages. As these tests demonstrate, increasing an endpoint's maximum number of outstanding messages allows the endpoint to inject larger bursts without overhead penalties. For the case where no injection policing is performed, injection overhead does not increase until burst size is larger than 5,000 null-length messages. At this point the NI buffers become saturated and the host must wait for space to become available in the NIs. As these tests demonstrate, a NI-based flow control scheme allows buffer space in the system to be utilized in a dynamic manner. This trait allows endpoints to inject a large number of messages with minimal amount of overhead for each message.

4.3 Logical Channels

In resource-rich cluster computers the communication library must be designed to allow multiple communication endpoints in a host to interact with the network through a single NI. In GRIM this task is performed through the use of NI-based logical channels. A logical channel in this context refers to a set of data structures housed in NI memory for facilitating message transfers between an individual endpoint and the NI. The NI is equipped with a logical channel for each endpoint in the host and therefore the NI must coherently transfer data between its collection of logical channels and the physical network at runtime. The advantage of this approach is that each endpoint is provided with its own virtual communication interface for the network.

The use of multiple logical channels in the NI has had a significant impact on the design of GRIM's NI firmware. One of the more challenging tasks in this effort has been adapting the reliable transmission protocol used by pairs of NIs to operate with multiple logical channels. The approach taken in GRIM is to structure the reliable delivery mechanisms to operate at the logical channel level as opposed to simply the NI level. This approach can help prevent head-of-line blocking that impedes communication performance. Another area of GRIM's firmware that was influenced by the use of multiple logical channels is the manner in which in-flight messages are buffered by NIs. Because there is a limited amount of memory available for implementing logical channels, modern versions of GRIM consolidate all NI-level message buffering into the sending NI. Finally, using multiple logical channels results in an increased workload for the NI. Therefore the GRIM firmware was examined to determine which areas are affected the most by the use of logical channels. Performance tests were constructed to determine the maximum number of logical channels a NI could support under practical conditions.

4.3.1 Logical Channel Structure

A logical channel provides a virtual communication interface that an endpoint can use to interact with the network. Each logical channel is equipped with data structures in NI memory that are necessary for maintaining this interface. Figure 4.7 depicts the data structures employed for a logical channel in the GRIM communication library.

In GRIM the three components of a NI-based logical channel are as follows.

- **Message Queue**: Each logical channel provides a dedicated amount of buffer space known as the message queue for housing in-transit messages. An endpoint supplies the NI with a new data message by appending the queue with the message and notifying the NI of the update. Multiple queue styles have been employed in GRIM and are discussed in Section 4.2.2.

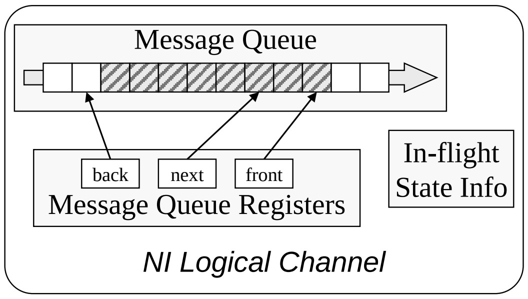

Figure 4.7: Each logical channel contains data structures necessary for providing a virtual communication interface.

- **Message Queue Registers**: A logical channel employs three registers for managing the capacity of its message queue. The first two of these registers are the front and back pointers which indicate the region of the queue that is currently occupied by in-flight messages. A third register provides a next pointer for the NI so that it can identify the next message to be transmitted in the queue.
- **In-flight State Information**: In addition to queuing data structures the NI must also maintain state information for each logical channel. This state includes sequencing information used by the NI's reliable transmission protocol and information about the endpoint that owns the logical channel.

A NI's logical channels are configured by the host when the system is initialized. Endpoints are connected to logical channels based on configuration information supplied to the communication library. In the current implementation the same logical channel provides both incoming and outgoing interfaces for an endpoint. While it is possible to use different logical channels to manage an endpoint's incoming and outgoing network interactions, doing so complicates the interface for the end user and is therefore avoided.

4.3.2 Message Sequencing with Multiple Logical Channels

The use of multiple logical channels affects the manner in which a reliable transmission protocol is implemented in the NI. The primary issue involves the manner in which pairs of NIs are synchronized to provide in-order delivery of messages from different logical channels. In the simplest approach logical channel information is ignored in the reliable transmission process. In this approach all logical channels are mapped on to a synchronous connection that exists between a pair of NIs. As described in Section 4.2.1, sequencing information is stored in two one-dimensional arrays: one for labeling outgoing messages and the other for verifying the order of incoming messages. The (outgoing/incoming) sequencing arrays are indexed by the value of the (destination/source) NI for the transmission. Sequence values in the arrays are incremented after every successful transmission. The downside of this approach is that congestion for one logical channel affects the performance of other logical channels. Since there is no way to distinguish which logical channel is congested at the destination, a request to retransmit a message forces the sending NI to perform rollback on all of its outgoing logical channels. This approach is undesirable, especially in resource-rich clusters where a host may have multiple endpoints that service incoming messages at different rates.

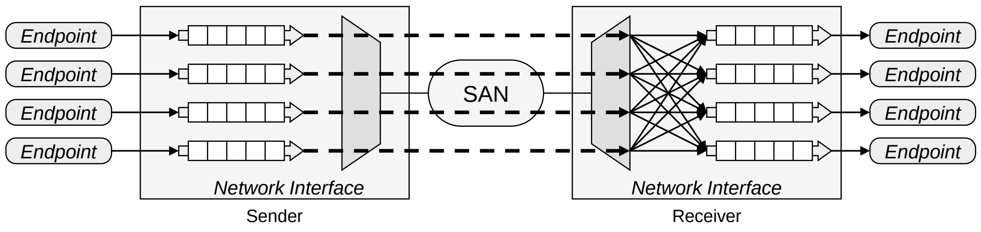

Figure 4.8: In the any-to-any approach, message sequencing is performed on messages based on the sending and receiving logical channels.

Another approach to implementing a reliable transmission protocol is to perform message sequencing on a *per logical channel* basis as opposed to a *per-NI* basis. In this approach each NI manages two three-dimensional arrays for sequencing values. These (outgoing/incoming) sequence arrays are indexed by the values of the source logical channel, the (destination/source) NI, and the destination logical channel. As depicted in Figure 4.8 this approach allows an any-to-any form of communication between sender and receiver logical channels. In this approach a message is retransmitted only if previous messages to the same logical channel were refused. The downside of this approach is that fetching sequence information is more time consuming and logical channels must maintain more state information. This approach is utilized in the current version of GRIM.

4.3.3 Distribution of NI Message Queues

Using multiple logical channels in the NI also affects the manner in which in-transit messages are buffered during the NI-NI communication process. In the ideal case it is desirable to provide buffering at both the sending and receiving NIs. Sending buffers allow the communication library to hide network congestion from the injecting endpoint. Likewise, a large receiving buffer for a NI can prevent the communication library from having to retransmit a message when a receiving endpoint is saturated. However, the issue with using multiple NI logical channels is that there is a finite amount of memory in the NI for housing the logical channels. Naturally, as the number of logical channels in the NI increases, the amount of buffer space provided to each logical channel decreases. Therefore it is necessary to consider how message buffering is performed in the NI in order to efficiently allocate the NI's buffer space.

In initial versions of GRIM, message buffering was provided at both the sending NI (outgoing queues) and receiving NI (incoming queues) as depicted in Figure 4.9. The intention of this approach is to split the NI's buffer space evenly between the sending and receiving tasks. Unfortunately there are at least three drawbacks to this approach. First, it was observed that the incoming message queues were used infrequently in the communication path. The characteristic can be attributed to the fact that the endpoints in the system commonly feature enough buffer space and processing power to match the rate at which messages arrived from the network. Second, using incoming message queues adds to the workload of the NI. While a cut-through path allows messages to bypass an incoming message queue when it is empty, the NI must still examine the incoming message queue when processing newly arrived messages in order to maintain ordered delivery. Finally, the allocation of incoming message queues decreased the amount of space available for outgoing

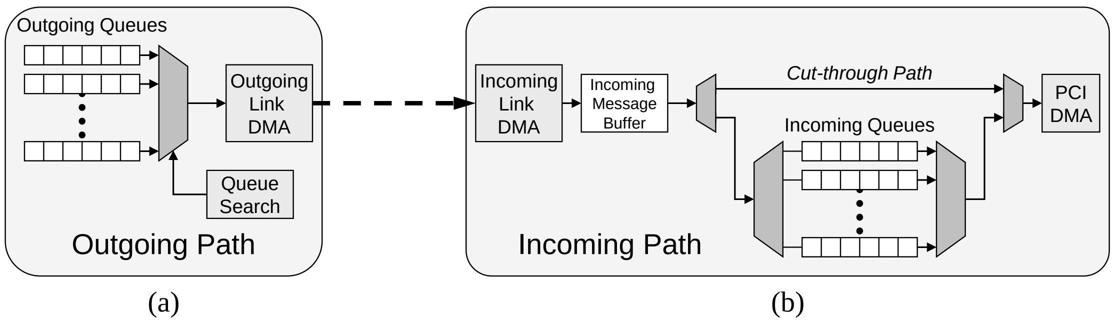

Figure 4.9: It is possible to buffer in-flight messages at both the (a) sending NI and the (b) receiving NI. A cut-though path at the receiver improves the performance of the receiving process.

message queues. This trait decreases the number of messages an endpoint can inject into the communication library at a time.

Based on these issues, GRIM was redesigned in a manner that only provides NI-level message buffering at the sending NI. In this approach a message that cannot be accepted by an endpoint due to a lack of endpoint buffer space is simply dropped by the NI and negatively acknowledged. Upon receipt of the NACK the sending NI automatically performs rollback on the appropriate outgoing message queue and retransmits the message at a later time. While this approach increases network load, Myrinet provides a considerable amount of bandwidth and retransmissions take pace only when a receiving endpoint is saturated. In addition to increasing the amount of buffer space available for housing outgoing messages, this approach reduces the overhead in receiving portion of the NI's firmware.

4.3.4 Number of NI Logical Channels

Utilizing multiple logical channels in the NI naturally results in an increased workload for the NI. Therefore it is beneficial to examine how the use of logical channels impacts the performance of the NI. A first step in this process is determining which portions of NI firmware are most affected by the use of logical channels. In GRIM's firmware the use of multiple logical channels has more of an impact on sending tasks than receiving tasks. In the sending portion of the NI's firmware the NI must inspect each outgoing logical channel to locate newly injected messages. Increasing the number of logical channels therefore increases the amount of time that a NI must spend searching for new messages to send. In contrast, the receiving process is not significantly affected by the use of multiple logical channels. This is because an incoming message contains all the information necessary for the NI to determine which incoming logical channel should be used to accept the message.

A second step in examining the impact of logical channels on NI performance is determining the maximum number of logical channels a NI can support under practical conditions. Given that the sending tasks of the NI are the region of NI firmware that is most affected, an experiment was constructed to determine how increasing the number of logical channels in the NI impedes performance. Specifically, this experiment is designed to measure the amount of time required for a NI to scan all of its outgoing logical channels for new messages. This scanning time is important because it can add delay to the total transmission time of an individual message. For example, a NI with 16 logical channels may have to scan

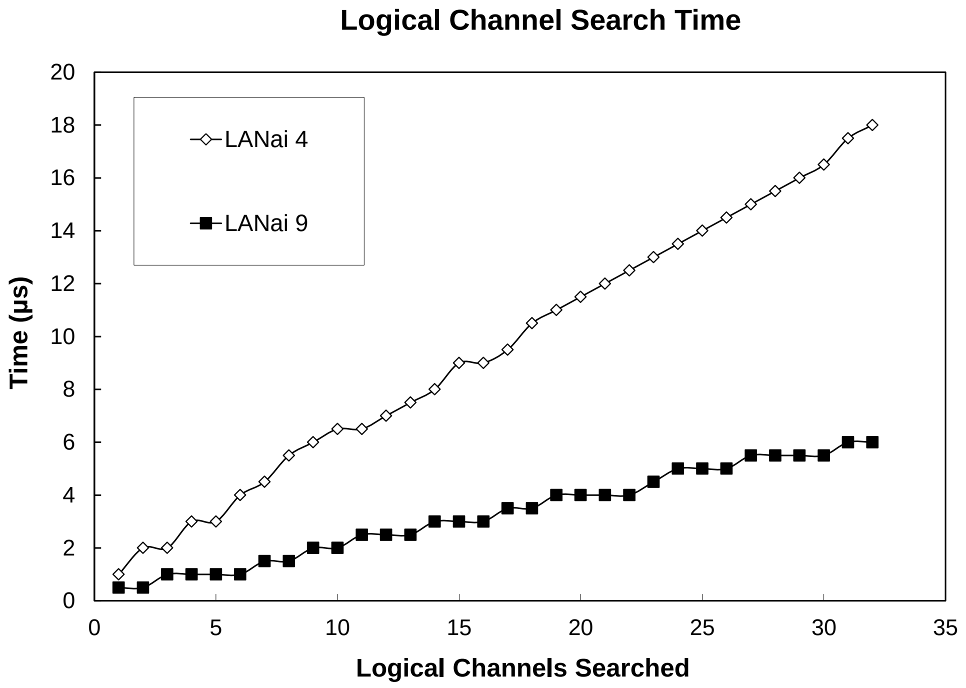

Figure 4.10: The amount of time required by the NI to search a fixed number of message queues for new messages.

15 empty logical channels before it can detect a message that was available all along in the last logical channel. Scanning time is also important because it consumes NI CPU cycles that could have been used to perform other NI tasks.

The results of the outgoing logical channel search experiment are presented in Figure 4.10. The test was performed using two versions of the Myrinet NI card, the LANai 4 and LANai 9. As expected the LANai 9's performance is roughly three times better than the LANai 4 due to architectural enhancements. When using a single logical channel the tests revealed that the LANai 4 and LANai 9 require 1 μs and 0.5 μs respectively for the NI to examine the logical channel. These times increase to 5.5 μs and 1.5 μs for 8 logical channels, 9 μs and 3 μs for 16 logical channels, and 18 μs and 6 μs for 32 logical channels. These results can be used to set a practical limit on the number of logical channels used in a NI. Given that most Myrinet communication libraries provide end-to-end latencies of approximately 10-20 μs, a conservative approach would dictate that in the worst case a NI would spend half of the potential delivery time searching for messages in the outgoing logical channels. Therefore, it is suggested that the LANai 4 and LANai 9 NIs use no more than 8 and 24 logical channels respectively.

4.4 Active Message Interface

The first of two APIs used in GRIM provides end users with an active message style programming interface. This interface is designed to be more robust than the original AM specification [27] due to the heterogeneity of communication endpoints used in resource-rich clusters. In GRIM, active message function handlers must be registered with a global server before they can be used by applications. In this process endpoints submit a string identifier for each function handler and are returned a unique integer identifier that any

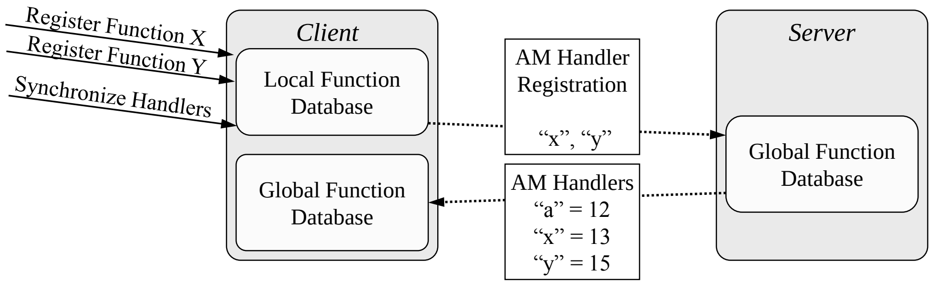

Figure 4.11: The active message API requires endpoints to register function handlers locally and then publish the information to a global database.

endpoint can use to reference the function. An active message contain all arguments necessary for an endpoint to process a message. Therefore GRIM endpoints employ a polling interface for detecting and processing incoming messages. The active message interface was extended with an overflow buffer to break the dependency between outgoing messages and incoming messages and thus avoid deadlock.

4.4.1 Active Message Handler Management

GRIM provides an infrastructure for dynamically managing active message function handlers used in the cluster. The challenge in this task is providing a means for endpoints to publish a list of available function handlers to the global context and have these handlers universally identified in a coherent manner. Since handler registration generally only takes place during system initialization, GRIM dedicates a single node in the cluster for providing global handler registration. In this approach endpoints submit a list of string identifiers for available function handlers to the server. The server correlates submissions and assigns unique integer identifiers for each string identifier. The list of mappings between string and integer identifiers is then published to all nodes in the cluster.

Figure 4.11 depicts an example of the active message handler registration process used in GRIM. The first step in this process is for an endpoint to locally identify its available function handlers using a local registration function. When the endpoint is ready for other endpoints in the cluster to begin using these handlers it executes a synchronization function. This function transmits the endpoint's table of available function handlers to the node in the cluster that manages the global database of active message handlers. The server processes the message by assigning new global handler identifiers for function handlers that have not yet been registered. The entire list of global function handlers is then transmitted back to the sender, where the data is stored in a global function handler database. At run time the endpoint consults this database to determine an integer identifier for a named function handler. If a node cannot perform the translation locally it can issue a request to the server to determine the proper integer identifier.

4.4.2 Polling Interface

Because of the nature of active messages, end applications do not need to provide specific reactions to the presence of incoming messages. Instead information included in the header of a message is used by the receiver to determine how to process the message. However it is still necessary for each endpoint to be equipped with mechanisms for detecting and

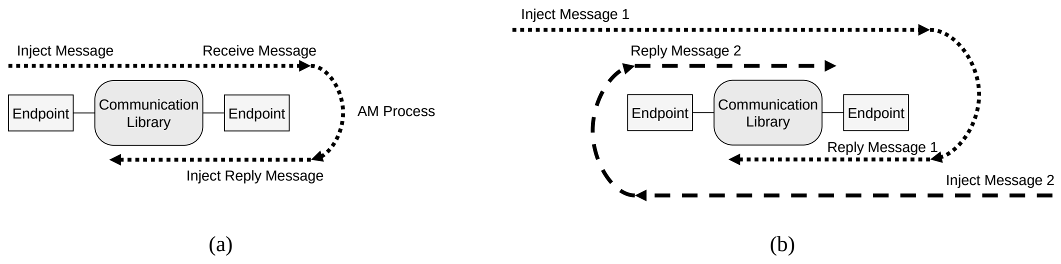

Figure 4.12: Example of deadlock at the application level. (a) The dataflow of messages for an active message handler that injects a reply message. (b) Application deadlock due to the simultaneous injection of two messages that require replies.

processing incoming messages. This functionality is accomplished in GRIM through the use of a polling interface. In this interface end applications must periodically invoke a polling function in order to guarantee that incoming messages are processed. Internally GRIM invokes polling operations whenever a user performs a GRIM call in order to guarantee that an application interacting with the network is at least performing a poll operation any time a network interaction is requested. For multi-threaded programs GRIM can be configured to dedicate a thread to periodically invoking a poll operation so that users do not need to explicitly poll the interface. Unfortunately this multi-threaded approach adds to the latency of communication.

4.4.3 Deadlock Avoidance for Message Handlers

A common problem in implementing an active message based system is that it is possible for deadlock to occur at the application level if precautions are not taken. Deadlock can occur because an active message arriving at a host can inject a reply message back into the network. The buffer space housing the incoming message cannot be freed until the handler completes and a handler that issues a reply cannot complete until buffer space is available in the network to inject the reply. As illustrated in Figure 4.12(a-b), this can result in a cyclic dependency between two applications when the network is congested. One option for removing this dependency is to utilize separate buffer space or separate networks for send and reply messages, and specify that a reply message cannot generate additional replies. This option is costly in terms of buffer space and limits the functionality of end applications.

A second option that is utilized in GRIM is to provide an overflow message queue at the host. If a handler must inject data back into the network and there is no room available for the outgoing message in the NI, buffer space is allocated in host memory to house the message. This memory serves as an overflow message queue, with additional injections appended to the queue until all overflow messages can be injected into the network. Since the host has a finite amount of memory, this approach does not guarantee that application deadlock due to message recycling will never occur. However with the large amount of virtual memory available to a host, this approach makes deadlock extremely unlikely and comes at little penalty to the common case.

4.5 Remote Memory Interface

The second programming interface provided by GRIM is for remote memory operations. This interface is designed to provide low-level mechanisms for manipulating and observing memory distributed throughout the cluster. For management purposes, GRIM requires that all remote memory operations utilize *registered memory*. Registered memory is memory allocated by GRIM that is guaranteed to always be available in physical memory. The remote memory interface utilizes virtual memory addresses to reference registered memory, and therefore requires mechanisms to translate a virtual address into a physical address that the NI can use to complete a remote memory operation. GRIM allows both remote memory reads and writes, and provides a simple notification mechanism to indicate that a transaction has completed. As a means of improving performance, GRIM also provides a special remote memory write operation that uses a physical address to reference registered memory. While there are basic rules that a user must follow when using the remote memory interface, the API provides a powerful means of managing distributed data in the cluster.

4.5.1 Registered Memory

The first step in using the remote memory API is obtaining a block of registered memory. Registered memory refers to a block of memory that has been allocated by the communication library and pinned so that the host's operating system does not attempt to relocate the block's physical pages. GRIM provides two interfaces for obtaining registered memory. The first interface obtains a block of memory in user space and then utilizes a system call to pin all of the pages of the allocation. While this approach can acquire large blocks of memory, its primary drawback is that the allocated memory is non-contiguous in the physical address space. This characteristic can result in reduced performance for NI interactions because when the NI accesses the memory, it must fragment its DMA operations into a series of page-sized transfers. Additionally, applications using this option must be given sufficient access privileges for invoking the system call that pins the memory.

GRIM provides a second interface for obtaining registered memory that uses a specially designed pinned memory management unit. This unit is a combination of both user- and kernel-level software and provides mechanisms for allocating large blocks of pinned memory that are contiguous in the physical address space. While the operating systems has a limited amount of contiguous memory, it is possible for this library to obtain multiple 4 MB regions from a host with 256 MB of system memory. There are two advantages to using this interface for obtaining registered memory. First, since this memory is contiguous, the NI can execute remote memory operations with a single DMA transfer. This feature decreases overhead and improves performance. Second, applications do not have to be given system privileges in this approach because a dedicated device driver performs the privileged task of interacting with the kernel's memory system.

4.5.2 Virtual Memory Translation in the NI

Remote memory operations are designed to use virtual memory addresses to reference a block of registered memory. Because the NI operates with physical memory addresses it is necessary for the NI to be equipped with mechanisms for translating a virtual address to a physical address. In GRIM the Myrinet device driver is designed to perform address translation for the NI on demand. In this process the NI DMAs an address translation request to a known location in the kernel's memory space and then triggers an interrupt

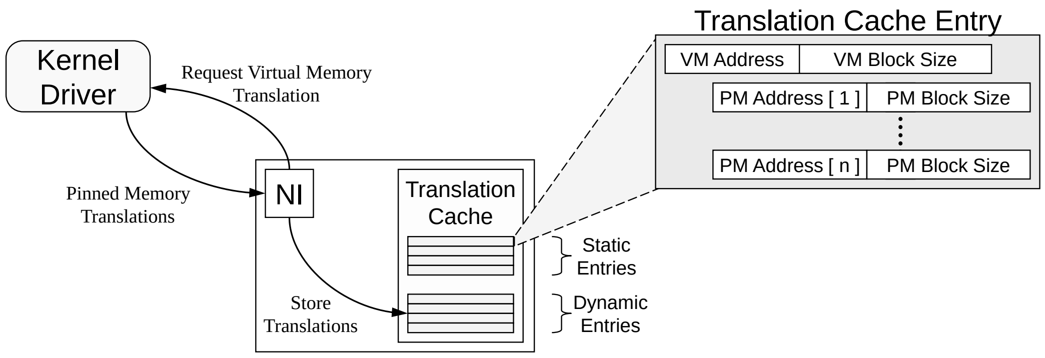

Figure 4.13: If a virtual memory translation is not available in the NI's translation cache, the Kernel must be consulted. An entry in the translation cache contains the size of a virtual memory block and a list of its physical memory regions.

signal to obtain the host's attention. The Myrinet driver handles the interrupt with an interrupt service routine that examines the request, performs the translation, and writes the resulting information back to the NI. Once equipped with a translation, the NI can process a remote memory operation.

Interrupt service routines are expensive operations for both the host CPU and the NI. Therefore it is beneficial if the NI is equipped with a means of caching address translations. As Figure 4.13 illustrates, GRIM's NI firmware is designed with a translation cache that is divided into two regions. The first region houses static translation entries that are programmed by the communication library when an application acquires a large block of contiguous registered memory. The second region of cache entries is for storing translations performed at runtime by the NI that were not satisfied by the first set of cache entries. This part of the cache is beneficial in situations where the first set of cache entries is full or when an application frequently accesses the same virtual address. Cache entries contain the virtual memory address for the registered block of memory, the size of the block, and the physical addresses and sizes of the pages that are used for housing the block of memory.

4.5.3 Remote Memory Writes (RM-V, RM-P)

GRIM provides two functions for performing remote memory writes. The first of these functions utilizes a virtual memory address to reference a remote node's registered memory and is referred to as an RM-V operation. At runtime the receiving NI must examine the arguments of the remote memory write operation, translate the virtual memory address, and perform the necessary DMAs to store the message's payload in the physical pages of the registered memory. Referencing the block of remote memory with a virtual address simplifies the interface for end users because local and remote endpoints can use the same memory pointer to reference a block of memory.

GRIM provides a second form of remote memory write referred to as an RM-P operation. An RM-P operation utilizes a physical address to reference a block of registered memory instead of a virtual address. RM-P operations are designed for experienced users that need to perform custom data transfers that take place efficiently. Since RM-P operations reference the destination's memory with a physical address, the receiving NI does not have to perform virtual memory translation to execute the message. Therefore RM-P messages are expected to have better performance than RM-V messages. GRIM provides a set of mechanisms for an end application to translate the virtual address of a local block of registered memory into a physical address. RM-P operations can also be utilized as an efficient means of updating the memory of a remote peripheral device.

4.5.4 Remote Memory Reads (RM-RV)

The second type of remote memory operation is designed to allow an endpoint to fetch data from a remote endpoint's memory. Remote memory reads operate using virtual addresses to reference registered memory at the sending and receiving endpoints and are referred to as RM-RV operations. RM-RV messages are utilized in GRIM as follows. First the sending endpoint injects an RM-RV message that contains references to (i) the receiver's memory that is to be fetched, (ii) the sender's memory where the results are to be stored, and (iii) the length of the transfer. When the message arrives at the receiving NI, address translation is performed and the requested data is fetched into a NI buffer. This data is then transmitted back to the sending NI in the form of an RM-RV reply message. Upon receiving this reply message the original NI translates the virtual address specified in the message, DMAs the message's payload to the address, and marks the original RM-RV message as acknowledged in its outgoing message queue.

4.5.5 Endpoint Notification for Remote Memory Operations

When utilizing a remote memory interface, a common operation is to transfer a block of data and then update a memory location in either the sending or receiving endpoint's address space to notify the endpoint that the transfer has completed. This notification operation can easily be performed using two remote memory operations, the first performing the transfer and the second performing the update. However, processing two instructions increases the workload of the communication library, which can degrade performance. Therefore remote memory operations in GRIM are equipped to provide a simple form of notification when the remote memory operation is completed by the NI. With this mechanism, users can specify the location of a single 32-bit word in registered memory that is updated when an operation completes. Notification information is stored in the user-defined arguments of a remote memory message. For remote writes the user can specify both the location of the variable to update as well as a 32-bit value to write to the variable. Remote memory reads carry more information in the message header and therefore only allow the user to specify the virtual address of a variable in the sender's address space that is to be updated. The NI writes a zero to this address when it finishes storing all of the fetched data.

4.5.6 Mixing Active Message and Remote Memory Operations

GRIM is designed to allow users to work with the active message and remote memory programming interfaces at the same time. This feature is possible because both interfaces are implemented as independent units that are layered on top of a system that reliable transfers messages between endpoints in the cluster. Since the delivery system forces each programming interface to adhere to a common message format, it is possible to mix traffic from different interfaces in the delivery system. The outgoing messages of different programming interfaces are merged when the endpoint injects the messages into the NI. For messages arriving from the network, the NI delivers the messages to the proper programming interface based on the type field of the messages. However, the active message and remote memory

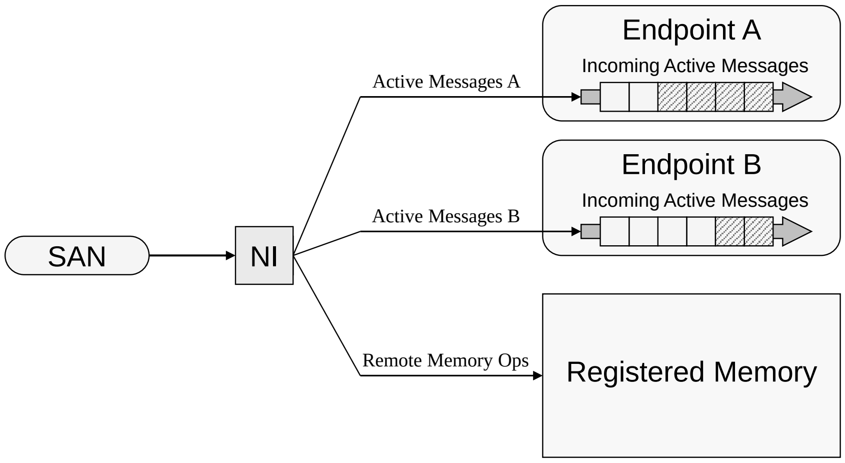

Figure 4.14: The data path for active messages provides extra message buffering before messages are processed compared to the remote memory data path.

programming interfaces have different strategies for buffering and processing messages arriving at the NI. Therefore it is necessary to specify the order in which incoming messages are processed when the two programming interfaces are used at the same time.

A message layer that implements in-order delivery between a pair of endpoints guarantees that messages are processed by the receiver in the same order that that they were injected by the sender. Unfortunately, differences in the manner that messages are buffered in the active message and remote memory programming interfaces make this guarantee undesirable when the two interfaces are used at the same time. As Figure 4.14 illustrates, the issue is that while remote memory messages are processed by the NI as soon as they arrive, active messages are placed in an additional endpoint-level buffer before they are processed. Therefore, in order to prevent remote memory messages from bypassing active messages, a strictly ordered system would require the NI to delay executing a remote memory message until the endpoint's active message queue is empty. This requirement impedes performance and negates the benefits of using the NI to process remote memory operations.

An alternative approach is to relax the requirement that the two programming interfaces are tightly synchronized in terms of processing order. In this approach messages are processed in the order in which they arrive at a programming interface, not the NI. An examination of the communication paths of GRIM reveals that this approach only violates out-of-order execution in one case: when an active message is followed by a remote memory operation. Because of the buffering of active messages in the endpoint, this approach can result in a remote memory operation being executed before preceding active messages are completed. However, all other uses of the communication library are guaranteed to take place in the order in which they are injected (AM followed by AM, RM followed by AM, and RM followed by RM). This approach is implemented in GRIM and requires users to be aware that remote memory operations may bypass previously injected active messages.

4.6 Summary

GRIM is a communication library designed for clusters that feature a high-performance Myrinet SAN. One of the key characteristics of GRIM is that core functionality is largely pushed into the NI cards. A NI-based reliable transmission protocol allows the NI to dynamically manage the transfer of data between NIs, and relies on an optimistic approach in order to decrease latency. Each NI is equipped with multiple logical channels in order to provide the various endpoints in the cluster with private communication interfaces to the network. The presence of multiple logical channels in the NI has resulted in changes in the way messages are buffered in the communication pipeline because each NI has limited a limited amount of on-card memory. GRIM simultaneously supports both active message and remote memory programming interfaces. These interfaces provide powerful programming abstractions that can be utilized in a flexible manner. This description of GRIM represents the core functionality of the communication library upon which extensions for resource-rich clusters are built upon.