CHAPTER VIII

MESSAGE LAYER EXTENSIONS

In addition to hardware extensibility, message layers for resource-rich cluster computers must also be designed to be support user-defined software extensions. These extensions allow users to customize their interactions with the message layer in order to implement functionality needed by applications. In GRIM, users can easily add new functionality to the core communication library at different levels of the message layer. At the network level, users can define new communication operations (e.g., multicast) by extending the message layer's NI firmware. At the endpoint level, users can implement extensions in a straightforward manner by constructing specialized active message function handlers.

This chapter deals with the issue of message layer extensibility in the context of application-related software extensions. Three specific software extensions have been constructed for GRIM to illustrate how new functionality can be easily incorporated into the library. First, multicast mechanisms have been added to the core library to allow users to easily transmit the same message to multiple receivers. These mechanisms result in a reduction in transmission overhead for the sender because the message is replicated in the network. These extensions demonstrate how users can insert new functionality at the NI level of the library. Second, fragmentation and reassembly mechanisms have been constructed for the active message and remote memory programming interfaces, as well as for multicast operations. These extensions illustrate how endpoint-level software can be added to the library to provide increased end-to-end performance. Finally, a reliable sockets emulation has been implemented using the active message programming interface. This emulation allows GRIM to be used as a replacement for the sockets library in legacy applications.

8.1 Multicast

A common operation utilized in parallel processing applications is multicast. Multicast is a form of communication where an endpoint transmits the same message to multiple receivers in the cluster. A subset of multicast is broadcast, where the list of receivers includes all endpoints in the cluster. Multicast messages are often used to distribute state information or provide synchronization among a number of cluster endpoints. In [56], researchers observed that these types of interactions have a strong impact on the overall performance of parallel processing applications. Therefore it is worthwhile to investigate means by which multicast can be performed efficiently in a communication library for resource-rich clusters.

While multicast operations can be implemented simply by layering this functionality on top of existing unicast mechanisms, doing so results in limited performance as the number of endpoints in a multicast distribution grows. Therefore it is beneficial to examine how the low-level mechanics of a communication library can be extended to support multicast more efficiently. Communication libraries supporting multicast typically perform the task of replicating multicast messages for a distribution tree in the NI [86]. Because these approaches recycle an incoming message back into the network, it is necessary for multicast mechanisms to be designed in a manner that prevents deadlock. The GRIM communication library has been extended with multicast support. In order to avoid deadlock GRIM employs

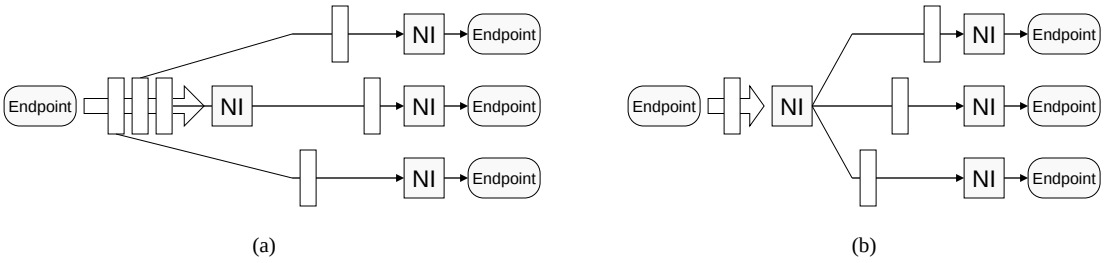

Figure 8.1: Replicating a multicast message can be performed by (a) the sending endpoint or (b) in the NI.

an ordering scheme on multicast trees that prevents cyclic dependencies between NI cards. In addition to preventing deadlock, this scheme utilizes a single message queue for recycled messages as opposed to two or more, resulting in an increased utilization of NI buffer space for multicast operations.

8.1.1 Multicast through NI Recycling

Multicast can be implemented on top of any unicast communication library simply by constructing endpoint software that transmits a separate copy of a multicast message to each receiving endpoint in a multicast distribution. An example of this approach is illustrated in Figure 8.1(a). While trivial to implement, this approach suffers from several drawbacks. First, this approach requires an endpoint to inject multiple copies of a message into the NI. As discussed in Chapter 5 the endpoint-to-NI transfers are the slowest part of the end-to-end communication pipeline. Therefore significant overheads may be accumulated by the endpoint for injecting multiple copies of the same message into the NI. Second, all transmissions for a multicast message must be serially transmitted through the sending endpoint's NI. Because the NI has limited amounts of buffer space for housing in-transit messages, it is likely that large multicast transmission will saturate the sending NI and delay message delivery. Finally, if endpoints are responsible for replicating a multicast message then every endpoint must be equipped with an up-to-date list of multicast receivers. This requirement makes updating a multicast distribution's membership list an expensive operation.

An alternative approach is to perform the task of replicating multicast messages in the communication library at the NI level. As illustrated in Figure 8.1(b), this approach is advantageous because a multicast message is only transferred once from the sending endpoint into the NI. This optimization reduces the load of the host's I/O system and greatly simplifies the amount of work an endpoint must perform to transmit a multicast message. In the context of resource-rich clusters, this approach is also beneficial because it is possible for all endpoints in the host to utilize multicast mechanisms because message replication is deferred to the NI.

Moving the task of replicating multicast messages into the NI requires consideration of how the NI should perform the task. Simply using the sending NI to transmit the multicast message to all receivers results in similar issues to the endpoint-based replication scheme: all multicast messages are serially transmitted by the sender and each NI must have knowledge of the entire list of multicast receivers. Therefore most NI-based multicast implementations are based on a distributed approach where the task of replicating messages is divided among

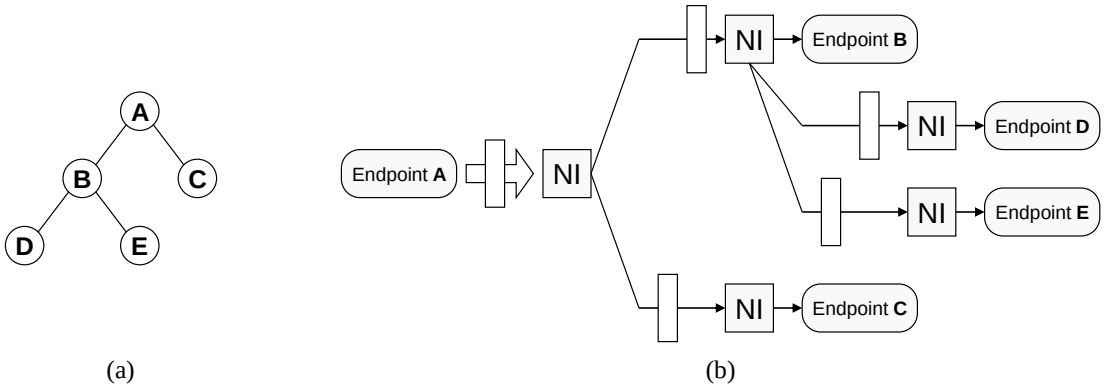

Figure 8.2: The task of replicating messages can be distributed among NIs through (a) constructing a distribution tree and (b) performing a limited number of multicast injections at each NI.

the NIs in the multicast group.

One approach to distributing the task of replicating multicast messages in the NIs is to organize the multicast group into a tree structure and then perform message replication in the individual NIs. An example of this approach is presented in Figure 8.2(a-b) with a five endpoint tree (a) that results in three NIs transmitting multicast messages to other NIs in the network (b). There are multiple advantages to distributing the task of message replication among NIs in the multicast group. First, multicast distribution can be accelerated because it is possible for multiple NIs to concurrently work on replicating a multicast message. Second, the workload for distributing messages is shared among all nodes in the group compared to requiring the sending NI to perform all of the work. Finally, individual NIs do not need to have knowledge of the entire multicast tree. Instead each node only needs to be equipped with the ids of its children and the root of the tree. This approach is sometimes referred to as recycling [63], as multicast messages are recycled back into the network during the distribution process.

8.1.2 Deadlock Issues in NI-Recycling Multicast

A hazard of using NI-recycling to perform multicast message distribution is that without precautions, it is possible for the network to become deadlocked. As illustrated in Figure 8.3(a), a NI that performs recycling takes an incoming multicast message and injects multiple copies of the message back into the network. Therefore if two or more NIs perform multicast recycling at the same time, it is possible for a cyclic dependency to be formed to between the NIs that can result in deadlock if the network becomes congested. An example of this type of condition is depicted in Figure 8.3(b-c). In this example two multicast trees have nodes 2 and 4 in common (b). Unfortunately these trees distribute messages in the reverse order, which leads to a cyclic loop between the two nodes (c). If the network becomes congested it is possible for the messages from these trees to reach a deadlock condition where incoming messages cannot be accepted because outgoing messages cannot be transmitted successfully to their destinations.

A common means of preventing this form of deadlock is to utilize the up*/down* routing algorithm first described for the Autonet network [82]. Up*/down* routing works with irregular topologies and is deadlock free [22]. In this approach a spanning tree graph is

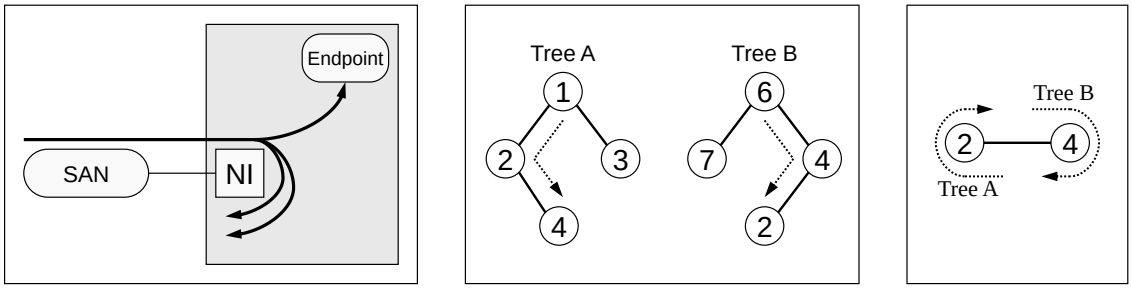

Figure 8.3: Replicating multicast messages in the NI results in a turn that could lead to a cyclic dependency loop.

constructed for the cluster with one node serving as the graph's root. Links in the graph are labeled with directions so that a node's "up" direction is towards the root and there are no directed loops in the graph (i.e., traversing the graph in the up direction never leads to a previously visited node). Nodes have two separate buffers for outgoing messages: one for upward-bound outgoing messages and another for downward-bound outgoing messages. At each node an incoming message can be ejected from the network, transferred from an incoming down link to an outgoing down buffer, or transferred from an incoming up link to either an up or down outgoing buffer. By preventing messages from traveling from a down link to an up link, cyclic dependencies are removed from the channel dependency graph resulting in deadlock freedom.

Up*/down* routing can be applied to prevent NI-recycling multicast mechanisms from reaching deadlock [53]. In this effort a directed acyclic graph is defined for all of the NIs in the cluster. This graph is fully connected because point-to-point networks allow any NI to directly communicate with any other NI. It is required that all multicast distribution trees be mapped onto the directed acyclic graph for the cluster. Each NI is equipped with separate up and down labeled logical channels for housing multicast messages that are recycled into the network. When a multicast message arrives at a NI from the network, the NI determines which multicast logical channel to recycle the message into based on the current direction of the message and the rules of up*/down* routing. Through these conditions cyclic dependencies between multicast buffers at different NIs are broken, allowing multicast transfers to take place without deadlock.

One of the conditions of using up*/down* routing for multicast is that the multicast distribution trees are arranged in a manner that agrees with the cluster's directed graph and routing rules. Some multicast distribution trees would violate these conditions and must be rearranged in order to prevent deadlock. An example of such a situation is presented in Figure 8.4(a-d) for a five node network that has the directed acyclic graph shown in (a). When the multicast distribution tree presented in (b) has its links labeled (c) using the cluster's directed acyclic graph (a), there is a routing problem at NI 4. In this tree NI 4 is unable to transmit messages from NI 1 to NIs 2 and 3 because doing so involves a transfer from a down directed link to an up directed link. Therefore the multicast delivery tree must be rearranged to a topology such as that presented in (d). This topology allows adheres to the routing rules and is guaranteed to not to create deadlock situations when used with other valid distribution trees.

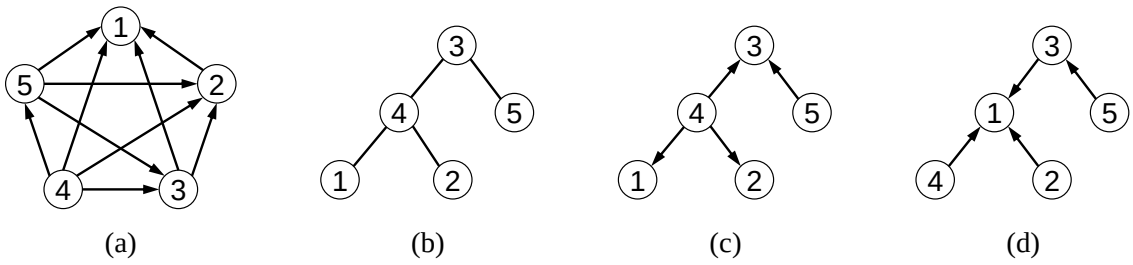

Figure 8.4: (a) A directed acyclic graph for a cluster's multicast transmissions. (b) A desired multicast distribution tree. (c) The desired multicast tree when labeled with link directions. (d) A reordering of the multicast tree that does not violate the up*/down* routing rules.

8.1.3 Multicast with a Single Recycle Queue

A criticism of up*/down* routing is that it requires the use of two separate message queues or logical channels for housing outgoing messages. While dynamic buffer space issues can be addressed through techniques such as escape channel routing [85], the primary issue in GRIM is that increasing the number of logical channels in the NI reduces the overall performance of the NI. Therefore it is beneficial to consider means by which the up*/down* routing algorithm can be applied in which multicast traffic can be housed in only a single outgoing logical channel. Reducing the number of logical channels used for multicast traffic to a single logical channel can effectively be accomplished by making restrictions on the manner in which multicast distribution trees are arranged in the cluster. Given that a two-channel up*/down* routing scheme already requires some multicast trees to be reordered, these conditions do not significantly impact end users.

The approach taken in GRIM to achieve deadlock-free multicast transmissions is to use a two-channel up*/down* routing scheme with the restriction that multicast trees are arranged in a manner that multicast messages always flow in the down direction of the cluster's directed acyclic graph. Because this system prevents messages from flowing in the up direction, the up logical channel can be removed from the NIs. One implementation of these conditions that is used in GRIM is as follows. A directed acyclic graph is constructed for the NIs in the cluster, where each NI has an up-directed link to every NI that has a smaller identification number in the cluster. An example of such a graph is presented in Figure 8.5. When multicast trees are constructed for the cluster, they are ordered in a manner such that a NI's ancestors in the tree are NIs with smaller id values and its descendants are NI's with larger id values. As a result multicast transmissions always propagate from a NI to a NI that has a larger id. Because data flowing from a NI with a smaller id to a NI with a larger id is always a down-traversal in the up*/down* graph, the up channel is not needed in this approach. The system however still operates under up*/down* routing rules and is therefore guaranteed to be deadlock free. Messages injected into a multicast tree must be transmitted to the root of the tree for transmission. However, since these messages are maintained in a separate logical channel at the sending NI, and incoming multicast messages cannot be recycled into this buffer, there is no dependency or possibility for deadlock.

The single recycle queue approach has both positive and negative characteristics for multicast transmissions. As discussed earlier the primary benefit of this approach is that

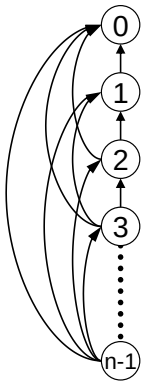

Figure 8.5: A directed acyclic graph for cluster nodes.

the sending NI needs to manage only a single message queue instead of two. Another benefit is that recycled messages are always placed in the same outgoing logical channel no matter where the destination is. In a two channel approach it is possible that the NI would have to store the same multicast message in both channels if the next two receivers in the tree had different link directions. As for drawbacks, this approach creates hotspots in the network. NIs with lower ids are more likely to be used for forwarding messages because they have more routing options than nodes with higher ids. Another negative aspect of this approach is that updating a multicast group is more complex in this approach because there are restrictions as to where a node can be placed in the distribution tree. Given that multicast tree updates are infrequent this factor is not a critical issue.

8.1.4 Implementation

The GRIM communication library has been extended to support a subscription-based form of multicast that is based on the preceding arguments. In this implementation the distribution of multicast messages to a group of subscribing endpoints is handled in the network by NIs that are equipped with a single, multicast recycling logical channel. The subscription nature of the implementation allows endpoints to dynamically join or leave a multicast group without significant overhead. The NIs for a multicast group are arranged in a binary tree topology, with each NI being responsible for distributing an incoming multicast message to (1) all the subscribing endpoints in the local host as well as (2) up to two other NIs in the cluster. In order to prevent deadlock, multicast messages may only flow from a NI to another NI that has a larger id number.

Applications utilize the multicast facilities of GRIM through a library of function calls. Each multicast group is statically labeled with a string name and dynamically assigned a globally unique integer identifier. One node in the cluster manages a database for translating multicast string names into runtime integer ids that can be referenced by all endpoints and NIs in the cluster. When an endpoint attempts to translate a string name that is not in the database, a new integer identifier is created and ownership of the multicast group is assigned to the endpoint requesting the translation. The owner of a multicast group serves as a central reference point for the multicast group and is responsible for dynamically managing the subscription list. Multicast management functions take place transparently in GRIM through the use of specially designed active messages. Therefore an endpoint

needing to communicate with a multicast group simply locates the multicast id for the group and injects a message that is marked with the id into the communication library. Multicast delivery is performed automatically by the communication library.

In order to perform the task of multicast distribution, individual NIs must be configured with two pieces of state information for each multicast tree. First, each NI is loaded with information that specifies the root NI for each multicast tree in the system. This information allows a message injected into the NI to be routed to the root of the multicast tree so that distribution can begin. If the originating NI is also the root of the multicast tree the message is simply moved from its outgoing logical channel into the NI's outgoing multicast logical channel when buffer space is available. The second piece of state data that NIs are loaded with is forwarding information. This information is used to determine which (if any) endpoints in the local host require a copy of an incoming multicast message, and which (if any) NIs in the cluster should be forwarded a copy of the message. When a NI is required to forward a message to other NIs, it inserts the incoming message into the NI's outgoing multicast logical channel once for all intended destinations and marks the message with appropriate forwarding information.

8.1.5 Multicast Group Updates

In the subscription-based form of multicast, endpoints can join or leave a multicast group dynamically. This operation is performed by transmitting a subscription update request message to the host-level endpoint in the cluster that manages a specific tree. After processing this request the endpoint determines the new multicast tree that needs to be constructed to satisfy the subscription update. The endpoint then transmits a special tree update message to the root of the new tree that contains the complete list of NIs that are part of the new tree. Upon receiving this message a NI will update its own local forwarding tables and then insert the message into its multicast logical channel for forwarding to its new children. These in-band update messages allow the multicast tree to be updated without involving the subscribing endpoints.

One of the challenges in implementing an system where multicast trees can be updated dynamically is correctness. In the ideal case all multicast messages are distributed to all of the endpoints that were part of a multicast group when the message was initially injected into the distribution tree. The use of in-band updates partially upholds this characteristic because when a NI updates its forwarding tables, it does not modify multicast messages that are already waiting for transmission in the outgoing multicast queue. Therefore forwarding changes are applied only to messages that follow the update message.

Another challenge in implementing multicast updates is preventing update messages from bypassing previously transmitted multicast messages for a tree. This condition is possible because update messages propagate through the cluster using the new multicast distribution tree instead of the old tree. For example, if linear tree A-B-C-D-E is being updated to linear tree A-D-F, it is possible that the update message will arrive at node D while multicast data is still queued in nodes B and C. Therefore GRIM implements a ticketing scheme [] that forces multicast message to be processed in the order they were injected. In this scheme the root NI labels each message with a ticket from a counter that is incremented after the transmission. NIs in the tree refuse to accept an incoming multicast message if its ticket value does not match the NI's expected value for the multicast tree. Therefore this system places strict ordering on multicast messages that prevents multicast messages for a tree from bypassing each other.

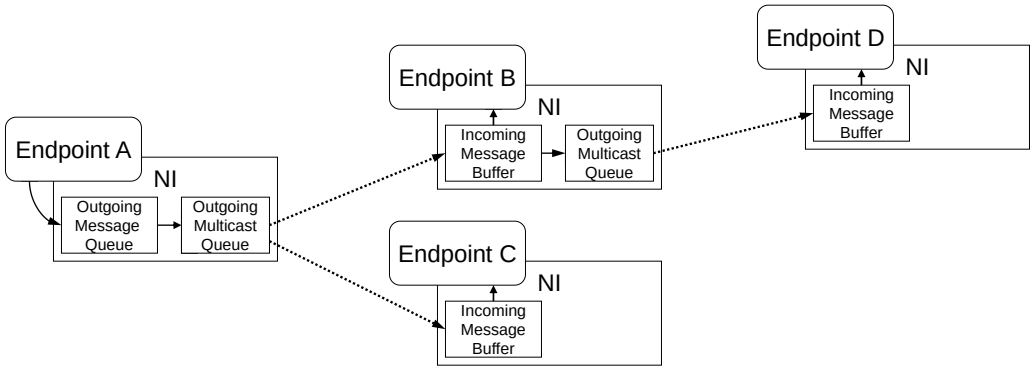

Figure 8.6: An example of the communication path for a multicast transmission.

8.1.6 Multicast Communication Path

As a first step in evaluating the performance of GRIM's multicast mechanisms, it is beneficial to examine the low-level details of the multicast communication path. An example of how data flows through the network components for a multicast tree is depicted in Figure 8.6. In this example endpoint A injects a new multicast message into its outgoing NI message queue. The NI is the root of the multicast tree and therefore after the NI detects the message it transfers it to the outgoing multicast queue. The NI then transmits the message to two other NIs where the message is relayed to endpoints B and C. While node C is a leaf in the distribution tree, node B must forward the message to node D. Therefore node B's NI copies the message into its outgoing multicast queue and transmits the message when the outgoing link becomes available. Node D receives the message and transfers it to its endpoint, completing the multicast operation. It is important to note that all network transactions in this process take place using per-hop reliable transmission mechanisms.

There are two observations that must be made bout the multicast communication path. First, at the injecting node a multicast message is buffered in an outgoing message queue before it is placed in the outgoing multicast queue. This buffering is necessary in order to guarantee that when endpoints inject multicast messages, the messages are properly inserted into the multicast queue. The downside of this approach is that compared to unicast messages, multicast messages always have a bit of added delay before they are transmitted into the network. A second observation of the multicast communication path is that there are multiple locations where a NI must copy a message from one NI buffer to another. While it is possible to perform some of these transfers in a cut-through manner, NIs often have limited bandwidth for local memory transfers. A test program was constructed to measure the memory copy performance of the Myrinet cards. This program revealed that the LANai 4 and 9 cards were only capable of transferring data at 19 and 66 MB/s respectively. Because of this poor performance it should be expected that NI-based recycling methods may not reach peak transfer levels observed in unicast procedures.

8.1.7 Multicast Performance

A series of benchmarks were constructed to observe the multicast performance of GRIM. In these tests multicast messages of variable sizes are transmitted to a set of receivers, which promptly transmit a null reply message back to the sender using unicast mechanisms. The sender measures the amount of time required to inject the multicast message and the amount

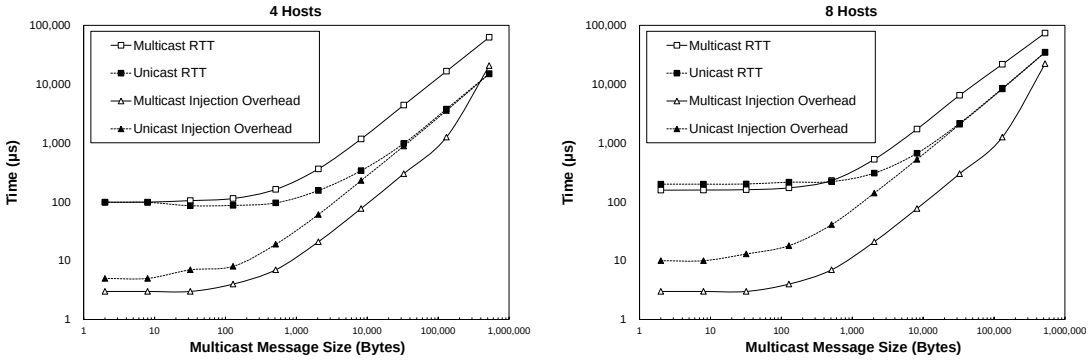

Figure 8.7: Performance of multicast and unicast messages for (a) four and (b) eight P4-1.7 GHz hosts using LANai 4 NI cards.

of time required to injected the message and receive replies from all destinations (i.e., the overall round-trip time). The benchmark uses two methods for transmitting a multicast message: the native multicast interface and a unicast system where the injecting endpoint injects multiple copies of the message using unicast calls.

The results of the benchmark are presented in Figure 8.7 for (a) four and (b) eight P4-1.7 GHz hosts. LANai 4 NI cards were used exclusively in these tests due to a lack of LANai 9 cards. The first observation of these measurements is that the multicast operations in general require less overhead to inject but have higher round-trip timings than the unicast operations. The reduction in injection overhead can be attributed to the fact that the multicast operation only has to inject a single message while the unicast operation must inject as many copies of the message as there are hosts receiving the message. The increased round trip latency for the multicast operation is due to the relatively high overhead involved in performing NI-recycling.

The timing experiments were repeated using multicast subscription sizes ranging from two to eight hosts. The round-trip timing measurements for the multicast transmission mechanisms are presented in Figure 8.8. In these tests, subscription sizes of 2-3 and 4-7 hosts were observed to converge in performance as the multicast message size was increased. This convergence can be attributed to the fact that the depths of the binary distribution trees for these subscription sizes were equal.

8.2 Message Fragmentation and Reassembly Mechanisms

In most packet-switching and wormhole-routed networks, messages are limited in sized to a fixed maximum transfer unit (MTU). Therefore it is beneficial to extend a communication library with functionality that allows a large message to be fragmented into a series of smaller transmissions that can be reassembled at the receiver. In addition to simplifying the communication interface for end users, these fragmentation and reassembly mechanisms can be used as a means of providing increased communication performance through message pipelining. GRIM provides a built-in mechanisms for fragmenting and reassembling active messages, remote memory messages, and multicast messages. For each category of message the fragmentation and reassembly mechanisms had to be designed to allow the messages to be transported reliably to the application in a transparent manner. Details of these procedures are provided in this section. Performance measurements are found in Chapter 5 for active messages and remote memory messages, as well as in the preceding section for

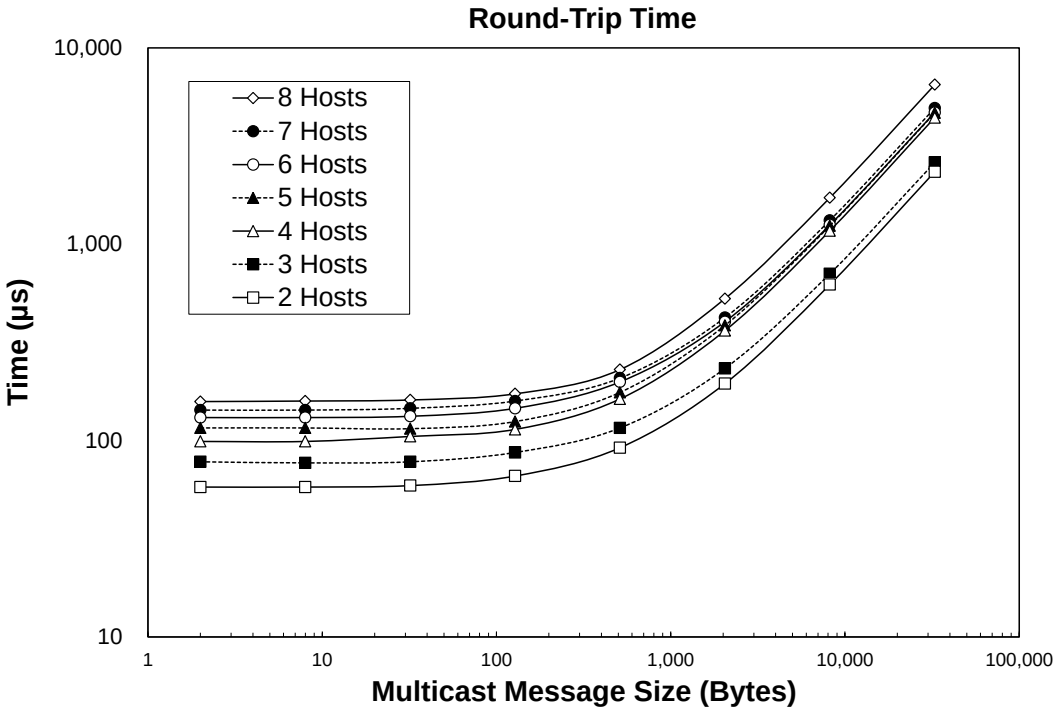

Figure 8.8: The measured round-trip times for different sized multicast groups.

multicast messages.

8.2.1 Active Message Fragmentation in GRIM

The first effort in providing fragmentation and reassembly procedures in GRIM is for active messages. Performing fragmentation on active messages is moderately challenging because of the manner in which the messages are processed by the receiver. In GRIM a receiver cannot execute an active message until all of its data has been transferred. Therefore the fragmentation procedures for active messages must be designed such that the receiver buffers a message's fragments and then executes the appropriate active message handler when all fragments have arrived. These procedures have been implemented in GRIM using a small number of active message handlers. The handlers use three types of active messages: one message to initialize the receiver, several messages to transfer the body of the message, and a finalization message to complete the transfer. The fragmentation and reassembly process using these messages is depicted in Figure 8.9.

The first message transmitted for a fragmented active message is an initialization message, which is designed to prepare the receiver for the incoming message fragments. The initialization handler allocates a reassembly buffer large enough to house all of the fragments and then provides the receiver with basic information about the original message, such as its active message arguments and function identifier. Following the initialization message is a series of body messages that contain fragments of the original message's data. The active message handler for a body message locates the reassembly buffer being used for the transfer and then copies the body message's payload into the proper offset of the buffer. The last message in the fragmentation process is a finalization message. The function handler for this message copies the last block of data into the reassembly buffer and then invokes the

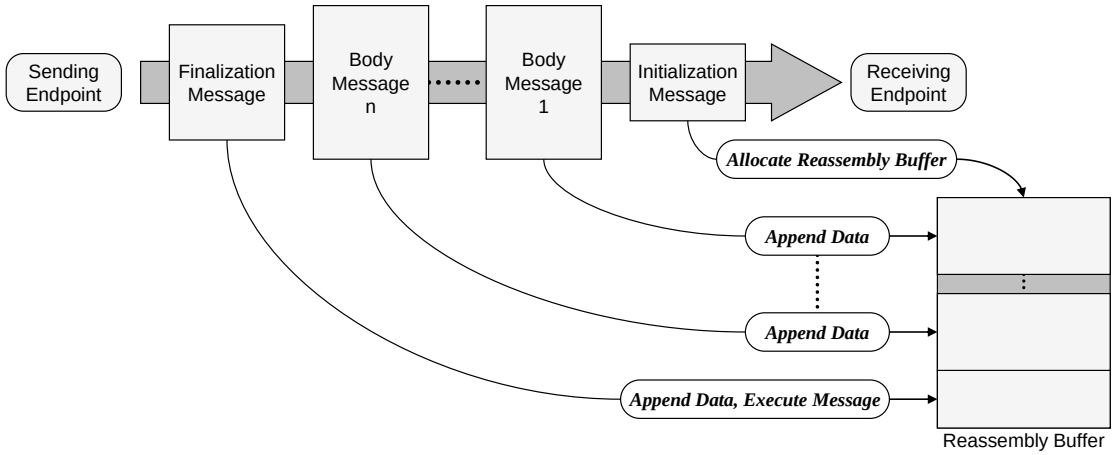

Figure 8.9: Fragmentation and reassembly of a large active message is performed by three types of active message handlers.

original message's active message handler. Once this operation completes, the finalization handler frees the reassembly buffer and clears the data structures used in reassembling the message.

There are multiple characteristics of this implementation that are beneficial to end users. From a performance perspective this approach is designed to hide overheads incurred by the fragmentations process. Most notably, the initialization message is designed to be small so that it can be transferred to the receiver quickly. This property allows the receiver to begin allocating space for the reassembly buffer while the first body message is being transferred. Another benefit of the fragmentation process is that it is implemented using existing active message communication mechanisms. Therefore fragmentation is easily layered on top of the system and operates in a transparent manner to end users. The use of active messages is also beneficial because end users can easily replace the fragmentation mechanisms with their own implementation by defining new active message handlers. This characteristic is particularly valuable when custom interactions with an endpoint are required by an application.

8.2.2 Remote Memory Message Fragmentation

It is much easier to implement fragmentation and reassembly procedures for remote memory operations due to the manner in which these messages are executed at the receiver. From an application programmer's perspective, remote memory operation simply transfers a block of memory from one endpoint to another and then optionally updates a user-space lock variable. Therefore a remote memory operation can easily be divided into a series of smaller transfers that are executed individually. If the user specifies that a lock update operation is to be performed at the end of a transfer, the last message in the series of fragmented messages can be configured to perform the operation. Because GRIM guarantees that remote memory messages are processed in order, the lock update takes place after all fragments for the transfer have been executed.

8.2.3 Multicast Message Fragmentation

Fragmentation and reassembly mechanisms for multicast messages in GRIM are similar to the mechanisms used for active messages. The key difference between these efforts is the manner in which endpoints maintain information about fragmented messages. In the active message procedures a unique id to reference a fragmented message is generated from the ids of the sending and receiving endpoints and a counter value. For multicast messages this reference value had to be modified because message fragments are transmitted to multiple receivers. Therefore fragmented multicast messages are references with an id generated by the sending endpoint id, the multicast tree id, and a counter value.

8.3 Protocol Emulation: A Sockets Interface

One of the challenges in constructing a communication library is addressing the issues involved in presenting users with a new API. Being human, application designers are naturally resistant to adopting new APIs. Additionally, new APIs generally prevent existing applications from being utilized with the communication library due to interface incompatibility. As there may be a significant amount of work involved in adapting existing applications to work with a new programming interface, it is desirable to provide mechanisms that allow a communication library to directly support a legacy API. This work is referred to as *protocol emulation* because the communication library provides a programming environment resembling that of a legacy API.

The GRIM communication library has been extended with functionality to support a basic emulation of the sockets API. In this emulation endpoints manage socket state in user-space and use a set of active message handlers to transfer socket data between endpoints. Macros are used to map socket API functions into the appropriate active message transactions. The emulation is able to detect whether a socket connection is for an internal cluster resource or an external host and provides the necessary connections in a transparent manner. Basic performance measurements have been made and suggest that while the sockets emulation is not as efficient as the SAN APIs, they are faster than traditional Ethernet-based mechanisms. These measurements indicate that legacy applications can directly use the communication library and benefit from its increased communication performance.

8.3.1 Sockets

The Berkeley sockets [24] interface is a well-understood mechanism for providing inter-process communication between two applications located on the same or different networked computers. At creation time a socket is specified as being either reliable (TCP based) or unreliable (UDP based). A reliable socket opens a bi-directional byte stream connection between two endpoints. While costly to establish, a reliable socket is suitable for long-term interactions between applications. An unreliable socket provides the user with a means of sending and receiving messages or datagrams between two applications. As the name suggests unreliable sockets leave the task of managing the reliable transport of data to the end application. While the extensions provided in GRIM are designed to only support reliable sockets, it is possible to implement an unreliable socket emulation in a similar manner.

Extending a communication library to support a reliable sockets interface can be beneficial for a number of reasons. First, since sockets-based programs are widely available, the application base for a communication library can be significantly increased through a

sockets emulation. Second, the performance of sockets-based applications may be enhanced through the use of a properly equipped SAN communication library. In addition to using a high-performance SAN instead of a LAN, there may be performance benefits in this approach because the socket operations are performed in user space instead of kernel space [10]. Finally, a sockets emulation for a SAN communication library allows an application to use the sockets API at the same time as the library's native API. Therefore users can construct applications that rely on the sockets API for routine endpoint interactions and then use the native SAN functions when increased performance is needed.

Because of the benefits of the benefits of a socket-based API, researchers have constructed sockets protocol emulations for existing communication libraries. One of the first and more notable of these efforts is the Fast Sockets project [80]. In this work researchers extended the AM communication library [27] to support a sockets API. The software would intercept calls made to the socket library and determine if the operations could instead be performed using the high-speed SAN and specially designed AM mechanisms. This work demonstrated that sockets calls could efficiently be layered on top of an existing SAN communication library. The researchers noted that while performance did not reach the peak levels offered by AM, there were significant gains over the traditional LAN mechanisms.

8.3.2 Planning a Reliable Sockets Emulation

There are at least three areas of development required to allow a SAN communication library to support an emulation of the sockets API. First, a conceptual model of the flow of data must be defined for the emulation. Socket data may be buffered at the receiving endpoint, the sending endpoint, or a combination of the two. While receiver-based buffering is more traditional, the other approaches may reduce the number of memory copies involved in transferring data in the emulation. Second, the communication library must be equipped with a set of functions for facilitating the emulation. These functions must be able to transport data between socket endpoints and maintain state information used in the emulation. Finally, wrapper functions must be constructed to allow the emulation to intercept calls to the sockets API. Wrapper functions translate socket functions into appropriate SAN transactions as well as convert traditional LAN information (i.e., IP addresses) into references that can be utilized with the SAN.

8.3.3 Implementation of a Reliable Sockets Emulation

The GRIM communication library has been extended with a software package that provides an emulation of the reliable sockets API. This software utilizes a small number of active message handlers for managing sockets and C-language macros to intercept a user application's socket operations. From a data transfer perspective this package is designed to buffer socket data at the receiving endpoint. This approach was selected for latency reasons, as buffering messages at the sender requires the receiver to perform a network fetch operation when an application attempts to receive data from the socket.

Internally each endpoint in the emulation maintains a list of open socket connections. An endpoint marks a port in this database as being available when an endpoint performs a socket operation for accepting a new connection. A connection is established when another endpoint in the cluster attempts to open a connection to the endpoint at an available port. Once connected two endpoints allocate data structures for buffering incoming socket data. When an endpoint injects data into the socket, an active message is used to transport the

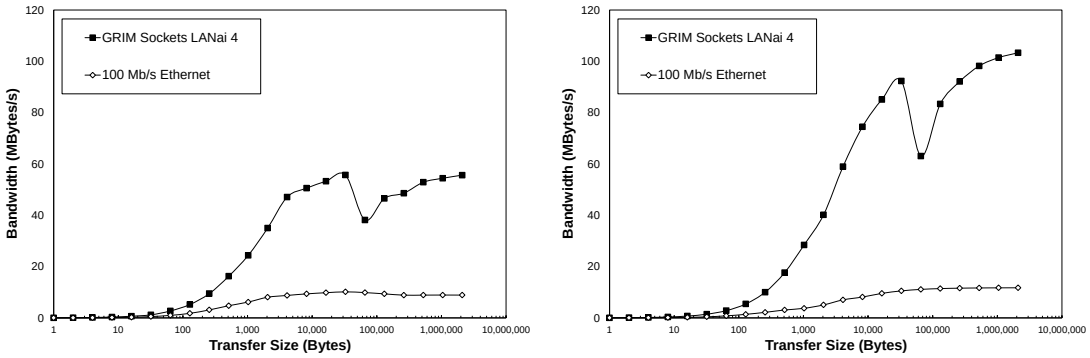

Figure 8.10: Performance of the GRIM sockets emulation using LANai 4 NI cards compared to 100 Mb/s Ethernet for (a) P3-550 MHz and (b) P4-1.7 GHz hosts.

data to the remote socket endpoint and append the data to its socket data buffer. The socket's read operation then examines the local buffer and extracts data as it becomes available.

One of the hardships in constructing a sockets interface is distinguishing between sockets used for internal SAN interactions and sockets used for other operations. For example, an application uses the same read and write operations to interact with a socket as it does a file. Therefore if read and write operations are intercepted by the emulation, the emulation must be able to determine whether to use the SAN library or to use the traditional library function. This task is performed in GRIM by intercepting all of the calls that manage file operations. When a new socket is opened, GRIM determines if the destination is a cluster resource and assigns these resources a specially marked file handler. The file handler returned for interactions with non-cluster resources is simply the file handler returned by the initialization operation. At runtime when a socket or file is accessed, GRIM can determine whether to use its SAN functionality simply by examining the file handler.

8.3.4 Performance

A benchmark program was constructed to measure the performance of the GRIM sockets emulation. This program was written using the traditional sockets API and therefore can be used with both a GRIM-based Myrinet network and a TCP-based 100 Mb/s Ethernet network. Selecting the communication library and network to use in the benchmarks is performed by setting a switch in the compile process. The benchmark program is designed to establish a connection between two hosts and transfer a block of data between the hosts in a round-trip fashion.

The results of the benchmark experiments are presented in Figure 8.10(a-b) and summarized in Table 8.1. As expected the GRIM sockets emulation outperformed TCP-based Ethernet for all transfer sizes due to the superior performance of the Myrinet SAN. GRIM provides roughly a third of the latency of TCP sockets and up to nearly nine times the bandwidth. The performance characteristics of the GRIM sockets API reveal that as messages become larger GRIM is able to provide better performance until a transfer size of approximately 64 KB. At this point GRIM begins fragmenting transmissions, resulting in a dip in performance. While performance begins to increase after this dip, it should be noted that the performance is not as high as that observed with the active message and remote memory interfaces. This characteristic can be attributed to the fact that the active

Table 8.1: Comparison of the performance of TCP and GRIM Sockets.

API	Network	P3-550 MHz		P4-1.7 GHz	
		Latency (μs)	**Bandwidth (MB/s)**	**Latency (μs)**	**Bandwidth (MB/s)**
TCP	100 Mb/s Ethernet	58.8	10.1	62.5	11.7
GRIM Sockets	LANai 4 Myrinet	22.7	55.6	22.2	103.3

message socket handlers allocate a new block of memory for every incoming socket fragment and add the block to a linked list. This differs from the fragmentation mechanisms in the active message interface where memory for a large transfer is allocated one time in advance and then filled with a series of transfers. However, the performance of this approach is reasonably high and is therefore beneficial as a means of improving the performance of legacy applications.