Efficient Multicast on Myrinet Using Link-Level Flow Control*

Raoul A.F. Bhoedjang Tim Rühl Henri E. Bal
Vrije Universiteit, Amsterdam, The Netherlands
{raoul, tim, bal}@cs.vu.nl

Abstract

This paper studies the implementation of efficient multicast protocols for Myrinet, a switched, wormhole-routed, Gigabit-per-second network technology. Since Myrinet does not support multicasting in hardware, multicast services must be implemented in software. We present a new, efficient, and reliable software multicast protocol that uses the network interface to efficiently forward multicast traffic. The new protocol is constructed on top of reliable, flow-controlled channels between pairs of network interfaces. We describe the design of the protocol and make a detailed comparison with a previous multicast protocol. We show that our protocol is simpler and scales better than the previous protocol. This claim is supported by extensive performance measurements on a 64-node Myrinet cluster.

1. Introduction

The importance of efficient multicast implementations is well understood. Multicasting occurs in many communication patterns, ranging from a straightforward broadcast to the more complicated all-to-all exchange. Message-passing systems like MPI [7] directly support such patterns by means of collective communication services and thus rely on an efficient multicast implementation.

We study the implementation of efficient multicast protocols for Myrinet, a switched, wormhole-routed, Gigabit-per-second network technology [3]. Today's wormhole-routed networks, including Myrinet, do not support reliable multicasting in hardware; multidestination wormhole routing is a hard problem and the subject of ongoing research [10, 16]. Meanwhile, multicast services must be implemented in software.

In this paper, we present an efficient and reliable multicast protocol for Myrinet. Like most software multicast schemes, our protocol forwards multicast packets along spanning trees, which allows packets to travel to different destinations in parallel. In most spanning-tree protocols packet forwarding is performed by *processors*: a processor receives a packet, delivers it to the application, and sends it to its children in the tree. Our protocol, in contrast, uses the *network interface* (NI) to forward multicast packets, thus avoiding expensive host-NI interactions. While several researchers have proposed NI-level multicast schemes [10, 11, 13, 17], few implementations exist. Moreover, several proposals ignore flow control, which is one of the key issues in implementing a reliable multicast. Even if the network hardware is reliable (as for Myrinet), flow control is needed to avoid receiver buffer overruns.

Our protocol has been implemented as part of a communication substrate called LFC (for Link-level Flow Control). LFC is targeted at developers of communication software for parallel systems and provides reliable, packet-based, point-to-point and multicast communication. Using LFC, we have ported MPI [7], Orca [2], and other systems to Myrinet.

LFC implements flow control at the *network interface* level. By implementing reliable, flow-controlled, communication channels between pairs of network interfaces the multicast protocol is greatly simplified. To avoid buffer deadlocks, LFC's basic multicast protocol restricts the shapes of multicast trees. By default, LFC uses binomial spanning trees, which give good overall performance.

This paper makes the following contributions. First, we show that NI-level flow control allows a simple and efficient multicast implementation. LFC avoids using centralized components, needs no special mechanism to deal with buffer overflow on NIs, and uses a single flow control scheme for unicast and multicast traffic. To illustrate these issues, we make a detailed comparison with a previous multicast implementation for Myrinet. This implementation, FM/MC [17], is one of the few NI-level multicast implementations that we are aware of and clearly demonstrated the performance advantage of implementing multicast at the NI level.

*This research is supported in part by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

Second, we show how LFC's basic multicast protocol can be extended with a deadlock recovery mechanism, such that arbitrary multicast trees can be used. Most store-and-forward multicast protocols restrict the topology of multicast trees to avoid buffer deadlocks. This is unfortunate, because it is well known that the optimal shape of a multicast tree depends on the communication behavior at the application level [4, 12, 13]. LFC assumes that deadlocks are infrequent and uses deadlock recovery instead of avoidance. When an NI suspects deadlock, it uses a slower, but deadlock-free protocol, which uses extra NI buffers to build escape paths [8].

Third, we evaluate the performance of LFC's multicast protocol on a Myrinet cluster. We show that LFC achieves both low latency (59 µs on 61 processors) and high throughput (13.3 MB/s on 61 processors). We study the performance impact of different multicast tree topologies and the deadlock recovery algorithm. Finally, we compare the performance of LFC and FM/MC.

The remainder of the paper is organized as follows. Section 2 describes the Myrinet architecture and summarizes FM/MC's multicast protocol. In Section 3, we describe LFC and its basic multicast protocol. This protocol is compared (qualitatively) to FM/MC's multicast protocol. Section 4 extends the basic protocol with a deadlock recovery scheme that removes restrictions on the multicast trees that are present in the basic protocol. In Section 5, we demonstrate that LFC's multicast protocol achieves good performance and in most cases outperforms FM/MC. Section 6 discusses related work and Section 7 concludes the paper.

2. Multicasting on Myrinet

Before describing LFC's multicast protocol, we summarize the Myrinet architecture and the FM/MC multicast protocol. FM/MC was implemented as an extension of Illinois Fast Messages 1.1 [15]. Both FM/MC and LFC implement multicast at the NI level, but their multicast protocols are very different, so it is interesting to compare these protocols and their performance.

2.1. Myrinet

Myrinet is a high-speed, wormhole-routed, switched LAN technology [3]. Myrinet network interfaces are attached to the host system's I/O bus and connect to each other via crossbar switches and high-speed links. Packets are wormhole-routed from one NI to another through a series of crossbar switches. Due to hardware flow control, Myrinet does not drop packets unless receiving NIs fail to drain the network.

Myrinet network interfaces have a programmable RISC processor (LANai), three DMA engines, and fast SRAM memory. The DMA engines are used to send to the network, receive from the network, and transfer data to and from host memory. All data transfers between the host and the network are staged through the NI's memory, which also holds the code and data for the program that controls the NI.

To avoid operating system overheads, the network interface's memory is mapped into the virtual address space of processes that use Myrinet. This allows the processor to move both data and commands to the NI's memory using programmed I/O (PIO). The NI, on the other hand, can access host memory only via DMA. Neither FM/MC nor LFC implements protection or device sharing between processes. These issues have been addressed in various other projects [5, 9].

2.2. The FM/MC multicast protocol

A detailed description of the FM/MC protocol is given in [17]. Below, we describe how FM/MC addresses the issues that arise in the design of a reliable multicast protocol.

Reliability and flow control. Reliability and flow control are closely related. In general, packet loss can have two different causes: unreliable hardware or receiver-side buffer overruns. The Myrinet hardware does not drop packets, but packets may be lost or corrupted when a receiving NI processes incoming packets more slowly than they arrive. Given a flow control scheme that stalls senders before receiver buffer overruns can occur, reliability can be obtained without retransmitting packets. Multicast flow control, however, is difficult, because buffer space is needed at multiple receivers, not just one.

FM/MC uses the following protocol. Each receiver allocates two types of buffers in host memory: one set of buffers is used for unicast messages, the other for multicast messages. An equal part of a host's unicast receive buffers is allocated to each sender. These unicast buffers are managed by a standard sliding window protocol that runs on the host. Multicast buffers, in contrast, are not partitioned statically among senders. Instead, a central credit manager, running on one of the NIs, keeps track of the number of free multicast buffers on each host. Before a process can multicast a message, it must obtain credits from the credit manager. To avoid the overhead of a credit request-reply pair for every multicast message, hosts can request multiple credits and cache them. Once consumed, credits are returned to the credit manager by means of a rotating token.

At the network interface level, no software flow control is present. Each NI contains a single receive queue for all inbound packets. When this queue fills up, FM/MC moves part of the receive queue to host memory to make space for inbound packets. Eventually, senders will run out of credits and the pressure on receiving NIs will drop; packets can then be copied back to NI memory and processed further. The idea is that this type of swapping to host memory will only occur under exceptional conditions, and it is assumed that swapping will not take too much time.

Deadlock. A multicast packet requires buffer space at all its destinations. This buffer space can be obtained before sending the packet or it can be obtained more dynamically, as the packet travels from one node in the spanning tree to another. The first approach, used by FM/MC, is conceptually simple, but requires a global protocol to reserve buffers at all destination nodes. The latter approach introduces the danger of buffer deadlocks. A sender may know that its multicast packet can reach the next node in the spanning tree, but does not know if the packet can travel further once it has reached that node.

In FM/MC, the credit and swapping mechanism avoid buffer deadlocks at the NI level: when NI buffers fill up, they are copied to host memory. The centralized credit mechanism guarantees that buffer space is available on the host. Also, since packets are not forwarded from host memory, there is no risk of buffer deadlock at the host level.

Multicast tree topology. Many different tree topologies have been described in the literature [4, 12, 13]. Depending on the type of multicast traffic generated by an application, one topology gives better performance than another. Shallow trees, for example, are best for low latency, because the path to each leaf in the multicast tree is short. Deep trees, on the other hand, give better throughput, because internal nodes need to forward packets fewer times, so they use less time per multicast. Our goal in this paper is not to invent or analyze a particular topology that is optimal in some sense. We simply note that since no single tree shape is optimal under all circumstances, it is desirable that a multicast protocol does not prohibit topologies that are appropriate for the application at hand.

FM/MC uses a binary tree (per sender) to forward multicast packets; it was established empirically that binary trees give good overall performance [17]. FM/MC's protocol, however, allows the use of arbitrary trees.

Division of labor between host and network interface. Programmable network interface processors typically are an order of magnitude slower than the host they connect to. Common sense therefore dictates that complicated protocol tasks be left to the host processor. Nevertheless, the programmability of an NI allows a number of simple, but effective optimizations.

FM/MC uses the NI for packet forwarding and credit management. NI-level packet forwarding is the key to its good performance. Both latency and throughput are improved because multicast packets need not travel to the host before they can be forwarded. Unicast flow control is performed entirely on the host. The multicast flow control protocol, however, uses one NI to manage credits and all NIs to forward the credit garbage collection token.

FM/MC's protocol has several disadvantages, some of which were identified in [10]. An important problem is that all credit traffic flows to and from the credit manager. This may increase multicast latencies and introduces a potential bottleneck. FM/MC is pessimistic in that it reserves buffers on all hosts before starting a multicast (but optimistic in that it never reserves NI buffers). Since only host buffers are subject to flow control, the NI may need to swap its receive buffers to host memory under heavy load. Finally, FM/MC uses two different flow control mechanisms: one for unicast and another for multicast. The two protocols cannot share each other's buffers.

3. LFC flow control and multicast

3.1. Link-level flow control

In contrast with other high-performance communication substrates (e.g., [15]), which minimize the amount of protocol code executed on the network interface, LFC carefully exploits Myrinet's programmable NI to implement flow control, to forward multicast traffic, and to reduce the overhead of network interrupts.

Unlike FM/MC, LFC implements peer-to-peer flow control at the network interface level. The flow control protocol guarantees that no NI will send or forward a packet to another NI before it knows that the receiving NI can store the packet.

LFC runs a straightforward sliding window protocol between each pair of network interfaces. Each NI dedicates part of its memory to receive buffers and assigns an equal number of buffers to each NI in the system. An NI that needs to transmit a packet to another NI may only do so when it has at least one *send credit* for the receiving NI. Each send credit for a particular NI corresponds to a receive buffer on that NI. Credits can be returned to their senders in two ways: by means of an explicit credit update message (when a low-water mark is reached); or by means of piggybacking (when a packet happens to travel back to the sender).

Figure 1 illustrates LFC's send path. The host copies message data into a free packet from the send buffer pool and enqueues a send descriptor in the send queue. The send descriptor contains the packet's destination and a reference to the packet. The NI, which polls the send queue, inspects the send descriptor. When credits are available for the specified destination, the descriptor is moved to the transmit queue; the packet will be transmitted when the descriptor reaches the head of this queue. When no send credits are available, the descriptor is added to a *blocked-sends queue* associated with the packet's destination. When credits flow back from some destination, descriptors are moved from that destination's blocked-sends queue to the transmit queue. Since the blocked-sends queue is always checked before other packets

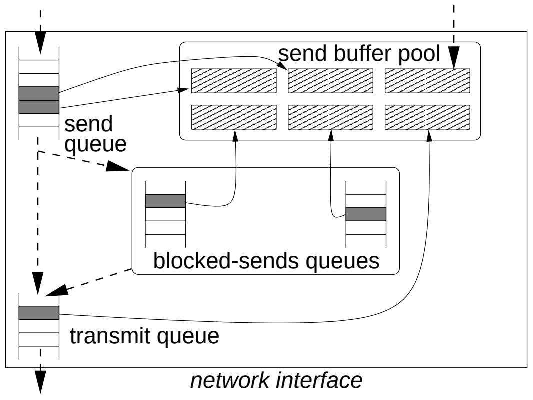


Figure 1. Send path and data structures.

can consume credits, packets are transmitted and received in FIFO order.

Receiving hosts are responsible for posting (via a shared queue) free host receive buffers to the NI. When no buffers are available, the NI simply does not copy packets to the host and does not release the packet's buffer space. Eventually, senders will be stalled and remain stalled until the host posts new buffers.

The sliding window protocol is essentially identical to the *host*-level protocol that Fast Messages (and FM/MC) uses for point-to-point messages. It is simple and efficient; LFC achieves similar point-to-point latencies as Fast Messages (10 µs, on the same hardware), which uses the host-level variant. Since the protocol preserves the reliability of the underlying hardware by avoiding receiver buffer overruns, senders need not buffer packets for retransmission, nor need they manage any timers. The main disadvantage of the protocol is that it statically partitions the available receive buffer space among all senders, so it needs more NI memory as the number of processors is increased.

3.2. Basic multicast protocol

LFC's multicast protocol is implemented on top of the reliable, NI-level protocol described above; it uses the same flow control mechanism and the same buffers. The protocol operates as follows. Each NI contains a table that specifies for each sender how packets from that sender must be forwarded along the multicast tree. Each sender has its own multicast tree. The NI uses the table to find the forwarding destinations of each multicast packet that it needs to send for its host or that arrives from the network. When the NI has to forward a packet, it creates a send descriptor for each forwarding destination. From then on, exactly the same procedure is followed as for a unicast packet. The packet will only be transmitted to a forwarding destination if credits are available for that destination; otherwise, the send descriptor is moved to the destination's blocked-sends queue.

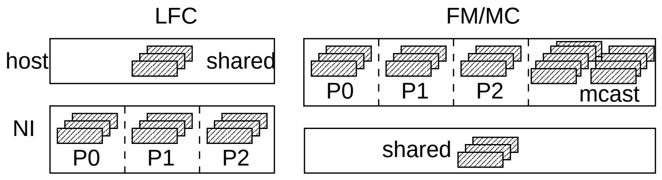


Figure 2. Buffering in LFC and FM/MC.

3.3. Protocol comparison

LFC and FM/MC offer similar multicast functionality: both provide a reliable, flow-controlled multicast primitive that preserves the (FIFO) ordering among multicast packets that originate from the same sender. Internally, however, LFC and FM/MC differ significantly in the ways they perform buffer management, flow control, and the tree topology they use. Below, we discuss these issues in more detail.

Figure 2 shows how LFC and FM/MC allocate receive buffers. At the network interface level, FM/MC uses a single queue of NI buffers for all inbound network packets, while LFC logically partitions its NI buffers among all senders. FM/MC's shared-buffer approach at the NI level is attractive when the amount of NI memory is small, because the number of buffers need not grow with the number of nodes in the system. Given a reasonable amount of memory on the NI and a modest number of nodes, however, partitioning buffers at this level poses no problems, and obviates the need for FM/MC's buffer swapping mechanism. On our 64-node system, LFC allocates 8 NI receive buffers (each 1 KB) per sender, which amounts to 0.5 MB per NI. With 8 credits, senders rarely run out of credits.

At the host level, the situation has almost been reversed. FM/MC allocates a fixed number of unicast buffers per sender and has another, separate class of multicast buffers. Senders are not given a fixed number of multicast buffers; instead, multicast buffer space must be requested from the credit manager. LFC, in contrast, does not partition its host buffers.

LFC and FM/MC also differ markedly in the way flow control is implemented. A multicast credit in FM/MC represents a buffer on *every* receiving host and is obtained by sending a request to the centralized credit manager. So, FM/MC always waits until every receiver has space before sending the next multicast packet(s). LFC, in contrast, does not distinguish between unicast and multicast credits and does not employ a centralized credit manager. The sender of a multicast packet does not need to wait for space on all receiving hosts or NIs, but only needs credits for its children in the multicast tree. No request messages are needed to obtain these credits: receivers know when senders are low on credits and send credit update messages without receiving explicit requests. In FM/MC, credits are not returned to senders, but to the credit manager.

The reason that multicast flow control in LFC is so much simpler, is that LFC decouples host buffers and NI buffers. In FM/MC, host buffers must act as a backing store for NI buffers. Reliability is only guaranteed when there are enough host buffers. To make sure enough host buffers are available, FM/MC uses the centralized credit manager. Multicast flow control cannot be integrated easily with unicast flow control, because the decision to forward a packet is taken on the NI, where the flow control information is not available. One solution is to let the host forward multicast packets, but this is expensive; the other solution is to move the flow control protocol to the NI, which is what LFC does.

Finally, we consider the types of multicast trees employed by both protocols. FM/MC uses binary trees, but its protocol is in no way tied to this topology. In contrast, LFC's multicast protocol (as described so far), is susceptible to deadlock when an arbitrary topology is used. When each sender's multicast tree is a linear chain, for example, deadlock can easily occur.

By default, LFC uses binomial spanning trees to forward multicast packets. Such trees restrict the routes that packets take to the routes prescribed by e-cube routing, which is deadlock-free [6]. In the next section, we show how LFC's multicast protocol can be extended with a deadlock recovery mechanism, thus allowing arbitrary multicast trees.

Summarizing, LFC's multicast protocol has important advantages over FM/MC. The most important advantage is its simplicity. A single flow control scheme is used for unicast and multicast traffic; there are no centralized components in the protocol; and there is no need to swap buffers back and forth between host and NI. The price to pay for this simplicity is the restriction on the tree topologies that can be used, but this restriction can be removed (see Section 4). With respect to performance, we note that LFC never sends credit requests and that credits are returned directly to their senders without waiting for a token to come by. Also, multicast packets can be forwarded as soon as credits are available for the next hop.

4. Deadlock recovery

To allow LFC to use arbitrary multicast trees, we have extended LFC's multicast protocol with a deadlock recovery mechanism. When an NI suspects deadlock, it switches to a slower, but deadlock-free protocol. This protocol requires $P-1$ extra buffers on each NI, where P is the number of NIs used by the application. Each NI owns one extra buffer on every other NI. The buffers owned by an NI form a dedicated *deadlock channel* on which only the owner may initiate communication. Deadlock channels are used to ensure forward progress when deadlock is suspected.

4.1. Deadlock detection

Deadlock recovery is triggered when a network interface suspects deadlock. LFC uses only local information to detect potential deadlocks, so no communication is needed to detect deadlocks. Multiple NIs can detect (and recover from) a deadlock simultaneously. While simultaneous recoveries do not interfere, they can be inefficient [14].

The detection algorithm is simple: each NI assumes deadlock as soon as it finds that it has run out of send credits for some destination that it needs to send a packet to. That is, nonempty blocked-sends queues signal deadlock immediately. Since a deadlock need not involve all NIs, we must monitor each individual queue; progress on some, but not all queues does not guarantee freedom of deadlock. Checking for a nonempty blocked-sends queue is efficient, but may signal deadlock sooner than a timeout-based mechanism. In Section 5 we evaluate the overhead of our detection mechanism.

When an NI has signaled a potential deadlock, it initiates a recovery action. No new recoveries are started while a previous recovery is still in progress. When a recovery terminates, the NI that initiated it will search for other nonempty blocked-sends queues and start a new recovery if it finds one. To avoid livelock, the blocked-sends queues are served in a round-robin fashion.

4.2. Deadlock recovery

The deadlock recovery algorithm must satisfy two requirements. Obviously, it must ensure forward progress. This is done using the extra deadlock buffers. Second, it must preserve the order among packets that are multicast on the same spanning tree.

The recovery protocol is illustrated in Figure 3. In this figure, vertices represent NIs and edges represent communication channels. Dashed edges represent channels where the parent has no credits for the child. When network interface R suspects deadlock, it selects a blocked packet p at the head of one of its blocked-sends queues and determines the multicast tree T that this packet is being forwarded on. The destination D of packet p forms the root of a *deadlocked subtree*, indicated with black vertices in Figure 3. The deadlocked subtree is a subtree of T in which parents do not have credits for their children and thus cannot forward packets of T. During a recovery, every node in the deadlocked subtree will be allowed to forward one packet to all its children in T.

We will now explain the protocol in more detail. First, R marks the selected packet p with a deadlock flag and trans-

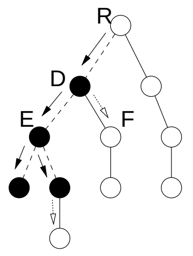


Figure 3. Deadlocked subtree.

mits it to *D*, where it will occupy one of the extra buffers. When an NI (e.g., *D*) receives a packet with this flag set, it belongs to the deadlocked subtree. Such an NI clears the flag and adds the packet to the blocked-sends queues for all its children in T. It then forwards, to each child, the first blocked packet (for that child) that belongs to T. This could be the packet just enqueued, but it could also be another packet sent earlier on T. Selecting the first packet in the blocked-sends queue preserves FIFOness between packets on the same tree. The NI transmits the packets to its children, setting the deadlock flag again only when no credits are available for some child. This way, the deadlocked subtree is extended. In Figure 3, *D* sets the deadlock flag in the packet sent to *E*, but not in the packet sent to *F*.

Using this algorithm, an NI that receives a packet with the deadlock flag set, will always forward, to each of its children in T, a packet of T (not necessarily the same packet that it received and not necessarily the same packet to all children). The key observation is that this will free at least one packet of T. The NI will tell its parent in T, by means of an acknowledgement packet, that it has freed a buffer. It will only do so, however, when it has received acknowledgements from all children that it added to the deadlocked subtree. That is, acknowledgements are propagated and merged in reverse direction along the deadlocked subtree. In Figure 3 one acknowledgement is sent for each black arrow, but in the opposite direction. When NI *R* receives an acknowledgement, it knows that its deadlock buffers on all NIs in the deadlocked subtree are free and it may initiate another recovery.

5. Performance evaluation

Below, we evaluate the performance of the basic protocol (without deadlock recovery) and the impact of deadlock recovery under conditions of light and heavy loads. In addition, we compare the performance of LFC and FM/MC. The fairly large number of nodes (64) in our cluster allows us to compare the scalability of LFC and FM/MC.

All experiments described below were performed on a cluster of 64 200 MHz Pentium Pros running BSD/OS 3.0. Each processor has 64 MB of DRAM, on-chip L1 caches (8 KB data and 8 KB instruction), and a unified L2 cache (256 KB). Each network interface has a 37.5 MHz LANai 4.1 processor and 1 MB of SRAM. The NIs are interconnected in a hypercube topology using sixteen 8-port switches and 2 $\times$ 1.28 Gbit/s cables. Each NI connects to its host's PCI bus (33 MHz, 32 bits wide).

At the sending side, both LFC and FM/MC move data from the host to the NI using programmed I/O. Receiving NIs use cache-coherent DMA transfers to move packets to host memory. Packets are received through polling and copied once after they have been DMAed. While LFC allows packets to be processed without being copied, most client systems make at least one copy, so we feel this extra copy makes the benchmarks more representative.

5.1. Performance of the basic multicast protocol

We first examine the performance of LFC's basic multicast protocol. For the following measurements, deadlock recovery has been disabled. We use 1 KB packets and allocate 8 send credits per sender per destination.

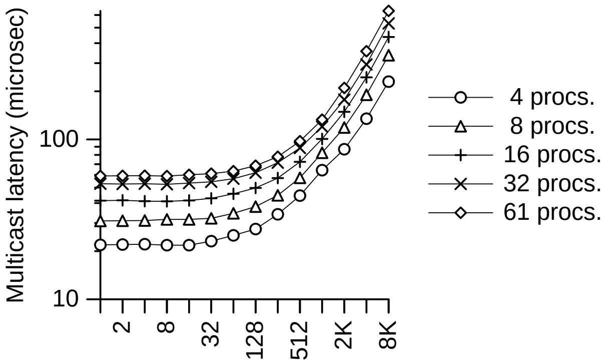
Figure 4 shows the latency (top) and throughput (bottom) for various message sizes and various numbers of processors. In both cases, we use binomial spanning trees. We report the latency observed by the last receiver of each (empty) multicast message. With 4 nodes, we obtain a latency of 22 μs; with 61 nodes[1], the latency is 59 μs.

Throughput is measured using a straightforward blast test. We report the throughput observed by the sender. With 4 nodes, the peak throughput is 42.9 MB/s; with 61 nodes, the peak throughput is 13.3 MB/s. Note that the 4-processor throughput curve declines when the message size reaches the size of the L2 cache (256 KB). At this point, the receiving processors, which copy data into a receive buffer, start to suffer from L2 capacity misses. The resulting memory traffic interferes with the NI's DMA transfers to host buffers. This does not occur for larger numbers of processors, because the incoming packets do not arrive at a sufficiently high rate to interfere with the host's memory copy.

5.2. Impact of deadlock recovery and tree shape

An interesting question is how much overhead the deadlock recovery scheme adds to the basic multicast protocol. We first examine the case in which no deadlock recovery is needed. In Figure 5, we show throughput results for the basic multicast protocol (BP) and the protocol with deadlock recovery (DR) for three packet sizes (512 bytes, 1 KB, and 2 KB). Receiver buffer space is kept constant at 0.5 MB. We use 61 processors and the same deadlock-free binomial spanning tree as before, so no true deadlocks can occur.

[1] At the time of writing, only 61 out of 64 processors were available.



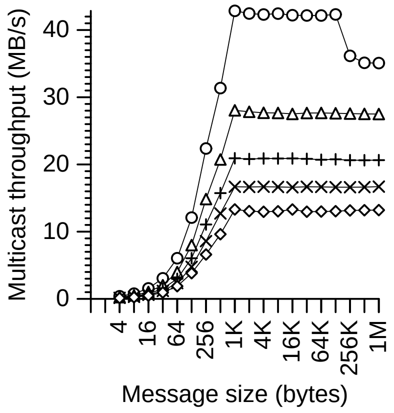


Figure 4. Multicast latency and throughput.

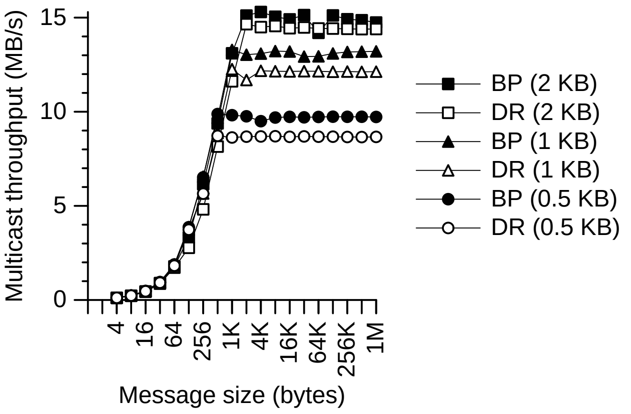


Figure 5. Multicast throughput with (DR) and without (BP) deadlock recovery.

Since our deadlock recovery scheme makes a local, conservative decision, however, it may still signal deadlock. In this experiment, processors that are near the root frequently initiate deadlock recovery. In the case of 1024-byte packets, for example, almost 25% of all multicasts results in a deadlock recovery action initiated by the root or a processor near the root. As the figure shows, however, this has only a modest impact on single-sender throughput. This is due to the small size of the deadlocked subtrees; in this example, an average of 1.3 (out of 60) packets per multicast is transmitted via a deadlock recovery channel.

Figure 6 (left) shows the impact of tree shape on single-sender multicast throughput. We compare binomial spanning trees, binary trees, and linear chains. We use a large number of processors (61), because that is when we expect the performance characteristics of different tree shapes to be most visible. In all cases, deadlock recovery was enabled, because with binary trees and chains, deadlocks can occur when multiple senders multicast concurrently.

First, we consider the impact of tree shape on single-sender throughput. In this case, no deadlocks can occur and we expect deep trees to perform well. This expectation is confirmed in Figure 6 (left). The linear chain outperforms both the binary and binomial spanning trees by a large margin; the difference between the binary tree and the binomial spanning tree is small. With 61 nodes, the binary tree is almost full; it has 6 levels and 30 nodes at the last level. The binomial spanning tree is broader and shallower: it has only 4 nodes at the last (6th) level and a maximum fanout of 6.

To test the behavior of the deadlock recovery scheme in the case that deadlocks *can* occur, we performed an all-to-all benchmark in which all senders broadcast simultaneously. We use the same multicast trees (binomial, binary, and chain) as in the single-sender case. The all-to-all benchmark is a worst-case scenario for our deadlock recovery scheme. First, true deadlocks are likely to occur for the trees that are not deadlock-free, especially for the chain. Second, since all processors multicast at the same time, the occupancy of the NIs increases because they need to forward many multicast packets. As a result, senders need to wait longer before their send credits are returned to them and deadlock will be triggered sooner.

Figure 6 (right) shows the per-sender throughput for all three multicast topologies (on 61 processors). First note that the per-sender throughput is much lower than in the single-sender case, due to contention for network resources (links, buffers, and NI processor cycles). As expected, the chain topology performs badly. The binomial spanning tree performs better than the binary tree.

Table 1 reports the frequency of deadlock recovery (as a fraction of the number of multicasts) and the average number of deadlock recovery packets per multicast. These statistics are for an all-to-all exchange of 64 KB messages on 61 processors. From these statistics it is clear why the chain performs badly: every multicast triggers a deadlock recovery action and all children in the multicast tree are without credits, so we are forced to always use the slow deadlock

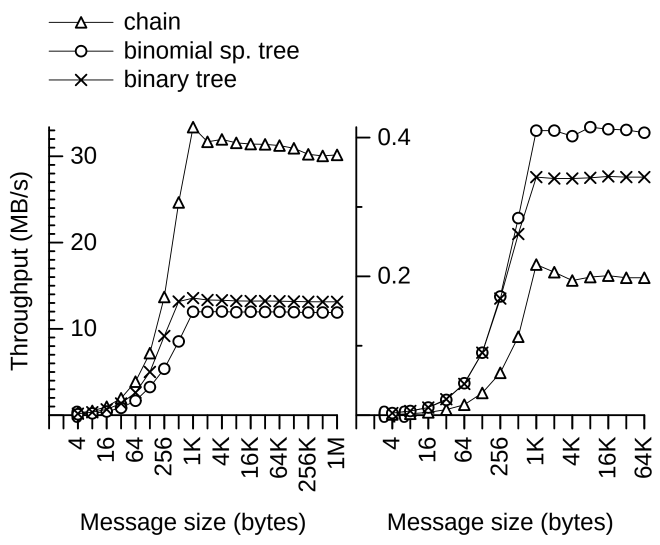


Figure 6. Multicast and all-to-all throughputs.

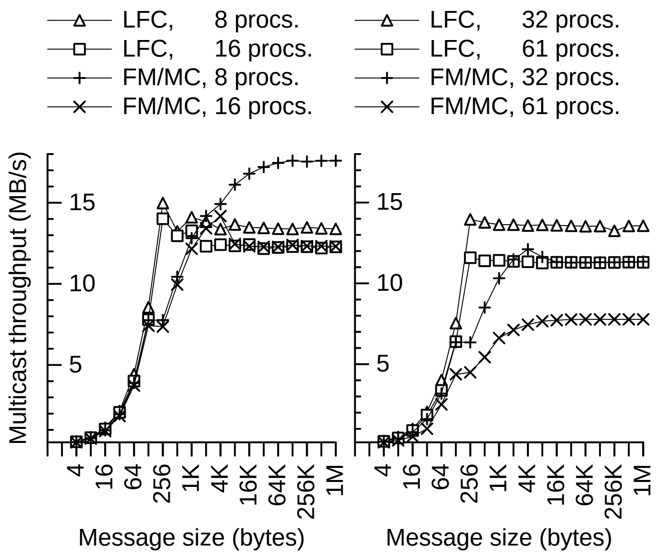


Figure 7. Multicast throughput comparison.

Tree shape	Recoveries per multicast	Avg. #deadlock recovery packets
binary tree	0.362	2.7
binomial sp. tree	0.853	3.5
chain	1.000	59.9

Table 1. Deadlock recovery statistics.

channels. With binary and binomial spanning trees deadlock recoveries also occur frequently, but here the size of the deadlocked subtrees is much smaller. Finally, note that binary trees trigger fewer and smaller recoveries than binomial spanning trees, but yet perform worse. Apparently, the performance difference is not caused by the deadlock recovery protocol.

Summarizing, we find that deadlock recovery is triggered frequently, but that its effect on performance is modest when the communication load is moderate and true deadlocks do not occur. To reduce the number of false alarms, we are considering a more refined, timeout-based, mechanism as in [14].

5.3. Comparison with FM/MC

Below, we compare LFC and FM/MC using the single-sender and all-to-all throughput benchmarks. We focus on throughput, because throughput benchmarks stress the flow control protocol. The difference in unicast and multicast latency obtained by LFC and FM/MC is small; usually, LFC's latencies are slightly lower. Before turning to the measurements, we discuss two factors that affect the performance of both systems: packet size and tree shape.

LFC uses variable-length packets with a maximum size of 1 KB, whereas FM/MC uses fixed-size, 256-byte packets. Fixed-size packets are slightly easier to process than variable-length packets, but have the disadvantage that they have to be relatively small to avoid a large increase in latency. Using small packets, however, often limits throughput. Due to its variable-length packets, LFC achieves both low latency and high throughput. On 61 processors, for example, LFC obtains a peak multicast throughput of 13.3 MB/s, versus 7.7 MB/s for FM/MC. To allow a fair comparison, we configured LFC to use 256-byte packets. This is a disadvantage for LFC, because LFC has slightly higher per-packet overheads which it can normally hide behind its larger packets.

To avoid differences in performance that result from differences in tree shape, we configured LFC to use binary trees, just like FM/MC. Since binary trees are not deadlock-free for LFC, we also enabled LFC's deadlock recovery mechanism. In a single-sender throughput benchmark, no deadlocks can occur, so we expect little overhead from the deadlock recovery mechanism. In the all-to-all benchmark, however, true deadlocks can occur, so we expect performance to suffer more noticeably.

Figure 7 shows single-sender throughputs on 8, 16, 32, and 61 processors. For 8 processors and large messages, FM/MC outperforms LFC, due to LFC's larger per-packet overheads. At 256 bytes, FM/MC begins to fragment messages, because FM/MC does not allow the user to completely fill the first packet of a message.[2] This explains the

[2]One word is used to store an active message handler.

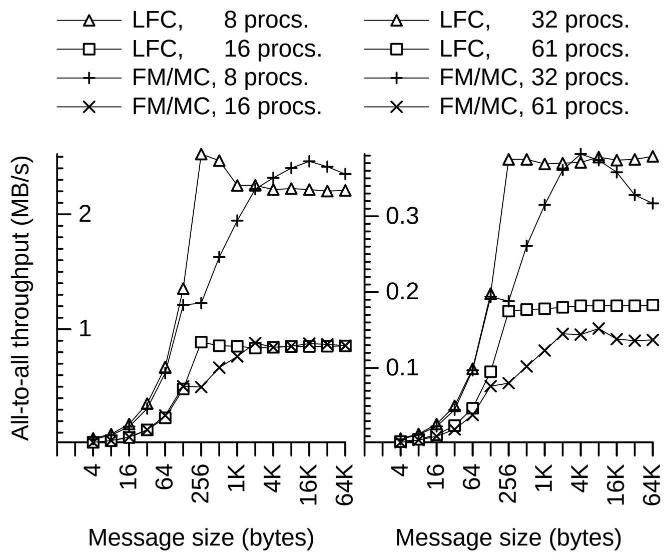


Figure 8. All-to-all throughput comparison.

step in FM/MC's throughput curves. Unlike LFC, FM/MC manages to increase its throughput while fragmenting messages; LFC always reaches its peak throughput exactly where fragmentation starts.

With 16 processors, FM/MC's peak throughput is better than LFC's, but for larger messages, FM/MC's throughput is the same as LFC's throughput. With more than 16 processors, LFC achieves much higher throughputs, which indicates that LFC's multicast protocol scales better than FM/MC's protocol.

Figure 8 shows the performance results for the all-to-all throughput benchmark on 8, 16, 32, and 61 processors. Again, for 8 processors and large messages, FM/MC outperforms LFC and at 16 processors performance is about equal. At 32 processors, however, LFC outperforms FM/MC and this continues to be the case for larger numbers of processors. In this benchmark, FM/MC suffers not only from its centralized credit manager, but also from its lack of NI-level flow control, which triggers the swapping mechanism described in Section 2.2 and degrades performance.

6. Related work

Using the network interface instead of the host to forward multicast traffic is not a new idea. Our NI-level protocol is original, however, in that it integrates unicast and multicast flow control and uses a deadlock recovery scheme to avoid routing restrictions.

In [11], the authors propose to exploit ATM network interfaces to implement collective communication operations, including multicast. In their symmetric broadcast protocol, NIs use an ATM multicast channel to forward messages to their children; multicast acknowledgements are also collected via the NIs. The sending host maintains a sliding window; it appears that a single window is used per broadcast group. LFC, in contrast, uses sliding windows between each pair of NIs. This allows LFC to integrate unicast and multicast flow control. The authors of [11] do not discuss unicast flow control and present simulation results only.

Gerla et al. [10] also discuss using NIs for multicast packet forwarding. Their scheme avoids deadlock by dividing receive buffers in two classes. The first class is used when messages travel to higher-numbered NIs, the second when going to lower-numbered NIs. This scheme requires buffer resources per multicast tree, which is problematic in a system with many small multicast groups.

Kesavan and Panda studied optimal multicast tree shapes for systems with programmable network interfaces [13]. They describe two packet-forwarding schemes, first-child-first-served and first-packet-first-served (FPFS), and show that the latter performs best. The paper does not discuss flow control. LFC integrates FPFS with its flow control scheme. As in FPFS, LFC forwards packets to all children as they arrive, but queues packets when no buffer space is available at the forwarding destination.

Several Myrinet-based systems employ different forms of link-level flow control. VMMC-2 [9], for example, treats the hardware as unreliable and saves transmitted packets in NI memory until they have been acknowledged by the receiving NI. In a multicast setting, the network interfaces would not only have to buffer packets that originate from their host, but also the packets that they must forward along the spanning tree.

AM-II [5], a protected cluster communication system, implements part of its flow control mechanism on the network interface. At that level AM-II uses multiple logical channels and runs an alternating bit protocol on each channel. Since AM-II does not statically partition NI buffers among senders, this scheme is augmented with a host-level sliding window protocol and NI-level retransmission.

LFC's multicast protocol can be implemented using any protocol that provides reliability at the NI level. Both VMMC-2 and AM-II provide this. Unlike VMMC-2 and AM-II, however, LFC assumes that the hardware is reliable and trades NI buffer space for a simpler flow control scheme between NIs.

Deadlock in spanning tree multicast protocols is often avoided by means of a deadlock-free routing algorithm. Most research in this area, however, applies to routing at the hardware level. Also, most protocols *avoid* deadlock by imposing routing restrictions [6] and this is the approach taken by LFC's basic multicast protocol. To allow different types of multicast trees that fit the communication pattern of an application, we have added a deadlock recovery scheme. This scheme was inspired by the DISHA protocol [1]. DISHA, however, is a deadlock recovery scheme for unicast worms, whereas our scheme recovers from buffer deadlock in a store-and-forward protocol for multicasting.

7. Conclusions

We have presented a simple, efficient multicast protocol for Myrinet. The protocol uses Myrinet's programmable network interfaces to efficiently forward multicast packets along a spanning tree. The key characteristic of the protocol is its use of hop-by-hop flow control at the network interface level. This is quite different from the flow control scheme employed by FM/MC, which uses end-to-end flow control and a centralized credit manager for buffer reservations. LFC's protocol is simpler and has two important advantages: multicast flow control is easily integrated with unicast flow control (e.g., buffers can be shared) and no centralized buffer management is needed for multicast buffers.

The main disadvantage of LFC's hop-by-hop flow control is the potential for buffer deadlocks. LFC's basic multicast protocol avoids such deadlocks by restricting the shape of multicast trees. We have shown that this restriction can be removed by means of a deadlock recovery scheme that requires extra buffer space on each NI. When no deadlocks occur, the overhead of this scheme is small.

LFC's basic multicast protocol performs well. Due to its use of variable-length instead of fixed-length packets, LFC achieves much higher throughputs than FM/MC. Even when we use equal packet sizes and the same multicast trees for both protocols, LFC almost always outperforms FM/MC for more than 16 processors. This performance gain is the result of LFC's NI-level flow control which does not use any centralized components and which avoids excessive pressure on network interface buffers.

Acknowledgements We thank Rutger Hofman, Thilo Kielmann, Koen Langendoen, Aske Plaat, Kees Verstoep, and the anonymous referees for their feedback on this work. We thank Andrew Chien and Scott Pakin for making FM available to us.

References

[1] K. Anjan and T. Pinkston. An Efficient, Fully Adaptive Deadlock Recovery Scheme: DISHA. In *Proc. of the 22nd Int. Symp. on Computer Architecture*, pages 201–210, Santa Margherita Ligure, Italy, June 1995.

[2] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Trans. on Computer Systems*, 16(1):1–40, Feb. 1998.

[3] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[4] J. Bruck, L. De Coster, N. Dewulf, C.-T. Ho, and R. Lauwereins. On the Design and Implementation of Broadcast and Global Combine Operations Using the Postal Model. *IEEE Trans. on Computers*, 7(3):256–265, Mar. 1996.

[5] B. Chun, A. Mainwaring, and D. Culler. Virtual Network Transport Protocols for Myrinet. In *Hot Interconnects'97*, Stanford, CA, Apr. 1997.

[6] W. Dally and C. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. on Computers*, 36(5):547–553, May 1987.

[7] J. Dongarra, S. Otto, M. Snir, and D. Walker. A Message Passing Standard for MPP and Workstations. *Communications of the ACM*, 39(7):84–90, July 1996.

[8] J. Duato. A Necessary and Sufficient Condition for Deadlock-Free Routing in Cut-Through and Store-and-Forward Networks. *IEEE Trans. on Parallel and Distributed Systems*, 7(8):841–854, Aug. 1996.

[9] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects'97*, Stanford, CA, Apr. 1997.

[10] M. Gerla, P. Palnati, and S. Walton. Multicasting Protocols for High-Speed, Wormhole-Routing Local Area Networks. In *Proc. of the SIGCOMM '96 Symposium*, pages 184–193, Stanford University, CA, Aug. 1996.

[11] Y. Huang and P. McKinley. Efficient Collective Operations with ATM Network Interface Support. In *Proc. of the 1996 Int. Conf. on Parallel Processing*, volume I, pages 34–43, Bloomingdale, IL, Aug. 1996.

[12] R. Karp, A. Sahay, E. Santos, and K. Schauser. Optimal Broadcast and Summation in the LogP Model. In *Proc. of the 1993 Symp. on Parallel Algorithms and Architectures*, pages 142–153, Velen, Germany, June 30–July 2 1993.

[13] R. Kesavan and D. Panda. Optimal Multicast with Packetization and Network Interface Support. In *Proc. of the 1997 Int. Conf. on Parallel Processing*, pages 370–377, Bloomingdale, IL, Aug. 1997.

[14] P. López, J. Martínez, and J. Duato. A Very Efficient Distributed Deadlock Detection Mechanism for Wormhole Networks. In *Proc. of the 4th Int. Symp. on High-Performance Computer Architecture*, pages 57–66, Las Vegas, NV, Feb. 1998.

[15] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, Dec. 1995.

[16] C. Stunkel, R. Sivaram, and D. Panda. Implementing Multidestination Worms in Switch-based Parallel Systems: Architectural Alternatives and their Impact. In *Proc. of the 24th Int. Symp. on Computer Architecture*, pages 50–61, Denver, CO, June 1997.

[17] K. Verstoep, K. Langendoen, and H. Bal. Efficient Reliable Multicast on Myrinet. In *Proc. of the 1996 Int. Conf. on Parallel Processing*, volume III, pages 156–165, Bloomingdale, IL, Aug. 1996.