IBRMP: a Reliable Multicast Protocol for InfiniBand

Qian Liu, Robert D. Russell
Department of Computer Science
University of New Hampshire
Durham, NH, 03824, USA
Email: qga2@unh.edu, rdr@unh.edu

***Abstract*—Modern distributed applications in high-performance computing (HPC) fields often need to disseminate data efficiently from one cluster to an arbitrary number of others by using multicast techniques. InfiniBand, with its high-throughput, low latency and low overhead communications, has been increasingly adopted as an HPC cluster interconnection. Although InfiniBand hardware multicast is efficient and scalable, it is based on Unreliable Datagrams (UD) which cannot guarantee reliable data distribution. This makes InfiniBand multicast not the best fit for modern distributed applications. This paper presents the design and implementation of a reliable multicast protocol for InfiniBand (IBRMP). IBRMP is based on InfiniBand unreliable hardware multicast, and utilizes InfiniBand Reliable Connection (RC) to guarantee data delivery. According to our experiments, IBRMP takes full advantage of InfiniBand multicast which reduces communication traffic significantly. In our testing environment, using IBRMP is up to five times faster than using only RC to disseminate data among a group of receivers. Compared to the MPI_Bcast, IBRMP is able to provide an equivalent low latency service in addition to its efficiency in large amount of data transmission.**

***Keywords*-HPC; Multicast; InfiniBand.**

I. Introduction

During the past few years, InfiniBand[1], a popular communications interconnect, has been widely adopted in data centers, TOP500 supercomputers[2], and other high-performance computing (HPC) fields. Because it is able to provide high throughput and low latency service, and is able to allow nodes to perform their own computation tasks while receiving data because of its asynchronous operation, InfiniBand is also gaining popularity in distributed applications, which usually disseminate data from one sender to a large group of receivers concurrently. For example, the University Corporation for Atmospheric Research (UCAR) Internet Data Distribution (IDD) project[3][4] distributes a great deal of meteorology data to more than 100 subscribers. Such an application generally requires a reliable multicast communication because multicast is an efficient approach to distribute data among clusters. Although it has benefits such as scalability and low overhead, InfiniBand multicast is built on unreliable communication which cannot guarantee ordering and message delivery. Lack of reliability makes the original InfiniBand multicast unfit for these distributed applications. Therefore, a reliable multicast communication for InfiniBand is necessary.

Most existing multicast protocols which provide reliable delivery utilize two popular approaches for end-to-end reliable multicasting: the sender-initiated approach and the receiver-initiated approach[5–10]. In the sender-initiated approach, the sender usually maintains a list which records the state of all its receivers to whom it has to send multicast packets and from whom it has to receive ACK packets which confirm the successful reception of multicast packets. The sender updates the list for the corresponding packet if it receives an ACK. Generally, the sender starts a timer for each packet, and uses the list as an index to retransmit a packet if the timer expires. In some cases, data retransmission is still based on multicast which cannot guarantee 100% delivery. Conversely, in the receiver-initiated approach each receiver has to notify the sender about any erroneous or missing packets, at which time the sender re-sends the requested packets to the receiver. This places the responsibility for ensuring reliable packet delivery on the receivers.

Several papers[5][6] have demonstrated that the receiver-initiated approach is more scalable than the sender-initiated approach. Also several optimization techniques have been applied to these two approaches. [6][11] propose to arrange nodes in a tree-based structure, dividing receivers into groups to achieve scalability with a large receiver set in order to solve the ACK implosion problem[12]. This shifts the burden of sending ACKs to a subset of receivers, which serve as subgroup leaders. Besides acting as normal receivers, these subgroup leaders, also called intermediate nodes, are responsible for collecting ACKs from other receivers in the same subgroup. However, if a group leader fails, a new group leader must be chosen to act in the same role. Information collected by the failed group leader has to be delivered to the new group leader, and other receivers remaining in the same subgroup have to establish new connections with the new group leader. This extra burden on receivers might degrade their local computation.

Other optimizations for a receiver-initiated approach are designed for peer-to-peer networks and a grid environment[7][13][14], in which a receiver asks for data from other receivers instead of the single sender node. In this approach, a mesh is built to transmit data among receiver nodes.

For a reliable multicast protocol designed for InfiniBand communication, it is not always acceptable to take advantage

978-1-4799-5860-3/14 $31.00 © 2014 IEEE
DOI 10.1109/HOTI.2014.24

IEEE computer society

of these improvements. For example, the disadvantage of implementing mesh sharing structure in IBRMP is that this approach adds extra processing overhead and a CPU burden on each receiver node. For instance, if there are N receivers and each receiver should choose M (M $\leq$ N) peer receiver nodes to randomly build its mesh, then a total of (N $\times$ M) / 2 queue pairs are built. And receivers have to constantly communicate with each other while processing incoming messages. Also, in some application environments such as IDD, nodes are receiving files selectively, which makes mesh sharing less effective.

Implementations of MPI_Bcast[15][16] have proposed approaches to achieve reliable multicast in InfiniBand. In[15], for example, a co-root scheme is used to solve the ACK implosion problem and also to address the drawbacks of the tree based ACK collection. Several co-roots help with both ACK collection and data retransmission. The load is more evenly distributed among the root and many co-roots. Another reliable multicast MPI implementation[16] depends on two-stage data transmission. In the first stage, the sender node broadcasts data on the multicast link. The second stage guarantees the data distribution. A virtual ring is built among all receiver nodes, and whenever a data packet is received successfully, each receiver sends it to its direct successor in a reliable way. However, IBRMP focuses on a network environment where receivers can join and leave the multicast group at any time, and they are not necessarily related with each other in any way. Therefore, it is impractical to maintain a tree or co-root hierarchy in IBRMP because each receiver has no knowledge about the other receivers, and it is expensive to find out who they are and maintain communications. Similarly, IBRMP cannot build a virtual ring among receivers, since it would be changing dynamically as receivers join and leave the multicast group.

Unlike previous work, IBRMP takes full advantage of InfiniBand hardware multicast, and utilizes InfiniBand Reliable Connection (RC) to guarantee data delivery in the receiver-initiated approach. Details of the IBRMP design are explained in Section III. Its performance evaluation is in Section IV.

II. Background

InfiniBand Architecture (IBA)[1] defines a switched fabric network for communication. It provides a management and processing infrastructure for inter-process communication and I/O. In an IBA network, nodes are connected to the fabric by a Channel Adapter (CA), which is similar to a Network Interface Card (NIC), but is equipped with a DMA engine and provides direct access interfaces for user-level applications to bypass Operating System intervention. Taking advantage of Remote Direct Memory Access (RDMA)[17] technology, InfiniBand is able to provide a high throughput, low latency, low overhead interconnect and is widely used in HPC fields and enterprise data centers. InfiniBand RDMA allows user level applications to move data directly between nodes without kernel intervention and extra data copying. It has two types of transport modes: Reliable Connection (RC) and Unreliable Datagram (UD), which are similar to TCP and UDP protocols respectively in TCP/IP networks. But transport modes provided in RDMA are message-oriented, and ordered delivery can be guaranteed in RC mode. Both RDMA transport modes support SEND/RECEIVE communication operations, in which both the sender and receiver must post a corresponding work request (WR) to its Queue Pair (QP) for data transfer. If there is no available WR posted for receiving, an incoming UD message will be dropped silently, while an incoming RC message will be resent. The retry time is based on negotiation at the connection establishment stage.

InfiniBand provides hardware support for multicast. A user is able to send a single message exactly once to a specific multicast address, and the CAs and switches will replicate it to multiple nodes in the same multicast group. Because multicast messages get duplicated at switches, communication traffic is reduced, and sender side overhead is minimized. Also, InfiniBand multicast latency is not sensitive to the number of nodes in the multicast group[15]. InfiniBand multicast uses UD transport mode to distribute messages, which means a single multicast SEND operation will consume one receive WR at each receiver side. Incoming messages get dropped silently if there is no available WR posted at receiver side. In InfiniBand multicast, the sender side never knows whether a message is successfully delivered to a receiver or not. Therefore, although it provides a scalable approach to distribute messages, InfiniBand multicast is an unreliable transport mechanism. It is not fit for distributing large amount of data among clusters that require reliable, in-order message delivery.

III. IBRMP Design

This section illustrates the architecture and design details of our InfiniBand reliable multicast protocol, which follows the principle concepts of VCMTP[8]. Unlike VCMTP[8], IBRMP has a simpler design. It doesn't maintain states in the sender and the receivers, nor does it perform any state transitions. In addition, it takes full advantage of InfiniBand features such as Shared Receive Queues (SRQ). The aim is to generate a minimal number of threads and occupy minimal system resources.

A. Architecture Overview

IBRMP is a message-driven based transport protocol, designed for file distribution among multiple nodes. At the server (sender) side, a file is segmented into blocks, which are transferred as messages via InfiniBand. Since message boundaries are preserved by the transport layer, it is easy for the client (receiver) side to detect missing messages by

comparing the message sequence number of the received message with that of the previously received message. If a receiver notices that a message lost, it sends a request message back to the sender side for retransmission.

IBRMP is provided as a library built on top of The OpenFabrics Enterprise Distribution (OFED)[18], as shown in Fig. 1. IBRMP avoids adding extra overhead by not introducing an extra layer which may add an extra wait-wakeup mechanism inside the library. The interaction between the IBRMP library and user-level application is easily done by using an IPC approach. Also, because it directly uses RDMA/IB verbs, this library can be integrated with OFED.

Fig. 2 illustrates the inner architectural overview of IBRMP. The IBRMP sender consists of multiple threads. The coordinator thread is in charge of the other threads. It starts the multicast thread (label (b)) after receiving the file transmission request from a user level application (label (a)). If multiple files are transferred simultaneously, multiple multicast threads are created, one for each file. In Fig. 2, three files are transferred at the same time, and their multicast messages are sent to the InfiniBand switch (label (c)). These messages will get duplicated at switch nodes and sent to each receiver node independently (label (d)) in the current multicast group, as shown in the dashed lines in Fig. 2. Because multicast is based on UD service, the multicast messages may get lost or dropped. If so, multicasting the same message many times may increase the percentage of successful reception of the missing message on receiver side, but it cannot provide a 100% guarantee. Therefore, another mechanism, which is able to guarantee message delivery, must be used for message retransmission.

In IBRMP, RC service is used for retransmission. On the sender side, an RC request listening thread is responsible for monitoring new reliable connection requests (label (e)), and it forwards such a request event to the coordinator thread (label (f)). After that, the coordinator thread creates one RC retransmission thread for each new connected receiver (label (g)). That RC thread is responsible for responding to

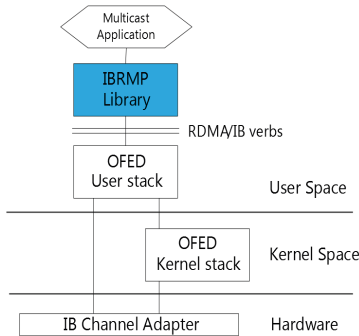


Figure. 1: IBRMP Implementation

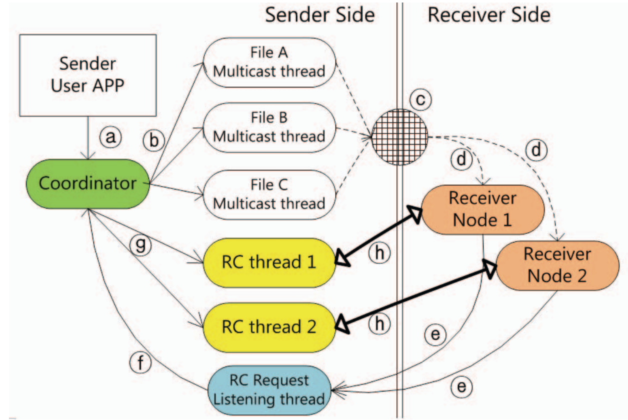


Figure. 2: IBRMP Overview

requests sent from that receiver node on the other side of the reliable connection, as label (h) shows in Fig. 2. The RC thread waits for a request in a wait-wakeup mechanism, it doesn't occupy CPU when waiting for requests. Unlike the multicast thread which multicasts messages to all receivers, each RC thread sends unicast messages directly to exactly one peer receiver by the SEND operation. IBRMP will implement RDMA_WRITE operation for the message retransmission in the future.

All RC threads on the sender side share one receive queue in their QPs. InfiniBand Shared Receive Queue (SRQ) allows sharing of receive buffers across multiple connections while maintaining good performance. This feature provides a scalable buffer management and the ability to handle a growing number of clients in an efficient manner for IBRMP.

Maintaining one RC thread for each receiver, however, would put too much burden on the sender side as the number of receivers increases. On the other hand, having only one RC thread which monitors all QPs for all receivers could also become overloaded. We provide a dynamic RC thread approach to solve this problem. A single RC thread services a limited number of receivers. We create the RC thread dynamically and assign that number of receivers to one RC thread before creating a new RC thread. This number can be controlled by the users calling our IBRMP library. Because users may have their own threads, and may want better control over the allocation of IBRMP library threads, the sender does not automatically create as many RC threads as the number of CPU cores available, nor does it automatically bind each RC thread to each core.

B. IBRMP message format and type

In IBRMP, the message formats and types are similar to those in VCMTP[8]. There are three types of messages transferred between sender and receiver: an ACK message, a request message, and a response message. The ACK

message is used for flow control on both sides. The request message and response message are totally different from ACK messages. The request messages are sent from the receiver side to the sender side. The response messages are sent from the sender side to the receiver side. Each message consists of a message header and a message payload, as shown in Fig. 3. Based on the different types of request message or response message, the length of the message payload may vary.

In the message header, the File ID indicates the unique ID belonging to the current transferring file. It is assigned by the sender and is used to uniquely identify each file sent. The fd field indicates the opened file descriptor of current transferring file. It is used to locate the corresponding file information on the sender side. The Sequence Number field indicates the start position of the data block within the file specified by the File ID field. Also, this field is used to detect message loss at the receiver side. The Length field indicates the number of bytes in the message payload. The Message Type field indicates the type of current message.

B.1 Messages Sent by Sender Node

Each time a new file transmission request arrives, its multicast thread multicasts a BOF (Begin of File) message which contains fundamental information about that file, and then multicasts all data blocks of this file. After multicasting a file, the multicast thread starts a timer and then notifies the coordinator thread, which notifies the RC thread to send an EOF (End of File) message to all its receivers. Therefore, even though a receiver may miss a multicast BOF and all multicast DATA messages of a file, it will receive the EOF message on its reliable connection and ask for retransmission of the BOF and all DATA messages. When the timer expires, the sender closes the corresponding file and releases all relevant resources. After this time, any retransmission requests for this file will not be processed.

1) BOF: A file's multicast BOF message is always transferred before any of its DATA messages. It notifies a receiver about the basic information for a file, including the total length and the file name. A retransmission requested BOF contains the same information but is transferred via RC.

2) DATA: This message contains the actual data of a file. The Sequence Number field in message header specifies the starting byte position of the data within the file.

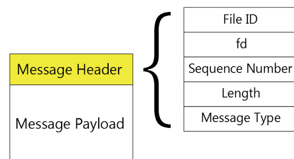


Figure. 3: Request and Response Message Header

3) EOF: After the multicast thread finishes multicasting a file, it sends an EOF message with the Length field set to the total bytes of the file. By doing so, each receiver can check file integrity and send corresponding request messages if necessary without waiting for the retransmitted BOF message. The EOF message is sent on each reliable connection to guarantee delivery.

4) FILE_TIMEOUT: On the sender side, if files were kept open indefinitely waiting for retransmission requests, subsequent file transmissions could be delayed due to resource limitations. Therefore, opened files must eventually be closed. We set an upper time limit for each opened file in order to close it. Accordingly, if a sender receives retransmission requests related to previously closed files, it will send this FILE_TIMEOUT message to the receiver. Also, closing a file after a period of time can guarantee robustness for receivers[8]. It gives much more time for slow receivers to get a whole file. The timeout mechanism is explained in subsection E.

B.2 Messages Sent by Receiver Node

On the receiver side, anytime a message received is not the one the receiver expects, a corresponding request message will be sent via RC back to the sender. The checking and processing logic of a received message is listed in Table I.

1) BOF_REQUEST: If a receiver detects a multicast BOF loss, it sends a BOF_REQUEST message to the sender. This could happen if the receiver starts receiving DATA messages or an EOF message of a new file without a previous BOF message for that file.

2) DATA_REQUEST: If the receiver detects one or more DATA messages missing, it sends this request to the sender. This could happen if the receiver receives a DATA message which has a noncontiguous Sequence Number when compared to the Sequence Number plus Length from the previously received DATA message. In such a request message, the Sequence Number field is used to specify the first missing byte, and the Length field is used to specify the number of missing bytes. For example, if a receiver receives a DATA message with Sequence Number 500 and Length

TABLE I: Receiver's Logic Checking and Processing of The First Received Message of a File

Message of a new file received at first	Receiver's actions
Multicast BOF	Notify user-level application that a new file is coming with its file information in BOF message
Multicast DATA	BOF of the new file was lost. Sends BOF_REQUEST message to the sender, and sends DATA_REQUEST message to the sender if necessary
EOF	BOF message and all DATA messages were lost, sends BOF_REQUEST and DATA_REQUEST to ask for the retransmission of the whole file

100, and then it receives a DATA message with Sequence Number 1000, it can determine that 400 bytes were lost and send a request message for data retransmission with a Sequence Number field of 600, and a Length field of 400.

3) TRANSFER_DONE: If the receiver finishes receiving a file, including all data and the EOF, it sends this message to the sender to report completion.

C. Flow Control And ACK Implosion

On RC, if there is no receive WR posted, any message sent by the other side will be retried a few times or forever based on the value negotiated at the beginning of the connection establishment. This retry occupies network and hardware resources and impacts transmission performance. In order to prevent sender and receivers from running out of available work requests, flow control must be applied.

A sender or receiver can only send response messages or request messages if it has available credits. The credit is the number of outstanding messages a sender or a receiver can send simultaneously without waiting for any confirmation. If credit is 0, any further posted WR should be put into a waitlist, waiting to be processed when credit is available again by receiving an ACK message, which is used to confirm a successful reception and processing of a request or response message. It gives the peer side corresponding credits to send further messages. Sending an ACK message doesn't require any credit.

Unlike a request message or response message, an ACK message can be designed as a 0 byte message, The immediate data field is used to acknowledge relevant message reception. On either side, sending an ACK message for each received request message or response message may improve the real-time performance of transportation, which means the credit can be restored quickly and the next message can be sent in time. However, a large number of ACK messages sent simultaneously may become an issue that impacts performance. Therefore, instead of sending an ACK for each received message, IBRMP batches the information that should be acknowledged in order to prevent possible ACK implosion, and then sends an "accumulated" ACK, which confirms all previously batched information.

D. Ghost Receivers

As illustrated before, an EOF message is sent to each receiver via the reliable connection in order to guarantee that no receiver would miss a whole file. Also, sending an EOF via the reliable connection can guarantee that no receiver would be hanging forever. If EOF were only sent via the multicast link, there is a possibility that a receiver may miss the last few DATA messages as well as the multicast EOF message, causing the receiver to wait forever, expecting new multicast messages because it doesn't know the multicast transmission is finished.

Although sending EOF message via RC can prevent these situations, it cannot prevent ghost receivers. That is, if a receiver joins the multicast group after the EOF has been sent, it will not realize it has missed the whole file because the EOF sending has finished before it joins the group. This situation is shown in Fig. 4. Initially receiver 1 and 2 are receiving data in the multicast group (label ①). After sending an EOF message, the sender waits for the completion notification from each receiver side, and these two receivers may be working with data retransmission. Then receiver 3 (label ②) joins the multicast group (label ③). To the sender side, there is one more receiver now in the multicast group, so the sender has to wait for all completion messages from all receivers before reporting file transmission status. However, the last joined receiver (receiver 3 in this case) didn't receive the EOF message which was sent before it joined, so it will not ask for any data retransmission or send a completion message to the sender. Therefore, even though other receivers (receiver 1 and 2 in this case) are finished, the sender cannot report transmission status until the file timeout. This "ghost receiver" blocks the sender from reporting transmission status before the file timeout.

To tackle this problem, the sender has to know all its "valid" receivers. Therefore, the sender can take a snapshot of the current multicast group (label ④ in Fig. 4) before sending the EOF message. This snapshot records all receivers of the current multicast group when the sender sends EOF messages. By doing so, if a receiver joins the multicast group after the start of EOF message sending, it has no impact on the sender because it is not in the snapshot. The sender can report any status change in time, and the newly joined receiver is just waiting for the subsequent transmission of the new files.

E. Timeout Mechanism

Unlike previous work[5][6][10][11], the IBRMP timeout is not related to each sent message, since IBRMP doesn't have to maintain status for each message. Also a large number of timers add extra overhead and would bring potential race conditions. Instead, the IBRMP timeout mechanism is

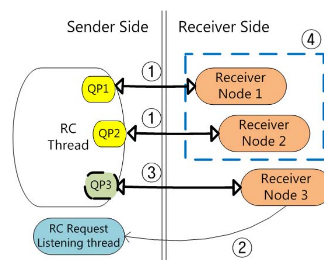


Figure. 4: Ghost Receiver Situation

related with the whole file. Also, since this timeout mechanism aims to guarantee robustness of receivers, it gives slow receivers much more time to ask for data retransmission.

Due to system resource limitations, opened files should be closed after a period of time in order to let new files transfer. If a file is closed, data in that file is no longer available, so any following retransmission requests related with that file would trigger the FILE_TIMEOUT message, which is sent back to the receiver. However, if there are many retransmission requests related with the same expired file, sending a FILE_TIMEOUT message for each request is redundant. Furthermore, the TIMEOUT message requires a credit. Therefore, sending such a message for each expired request would slow down the DATA retransmission.

In IBRMP, we selectively send the FILE_TIMEOUT message back to receivers via the reliable connection. This selective sending mechanism can guarantee that if requests from a receiver are no longer valid, that receiver will get a FILE_TIMEOUT message only once. Also, this mechanism can minimize overhead, and save credits as much as possible by not sending FILE_TIMEOUT messages to a receiver if it determines that receiver will get such a notification by other means.

On the sender side, the combination of the File ID field and the fd field in message header is used to determine whether a file expires or not. As illustrated before, File ID is the value which identifies a file uniquely. This value is incremented as each new file opened. The fd field is the file descriptor returned by the *open()* system call. There may be no upper limit for the File ID, but there is an upper limit for the number of opened files. Therefore, the complexity of locating a file is O(1) by using the combination of these two fields. Each time a sender multicasts messages or unicasts messages, it sets the fd field in the message header to the fd of current file. Each time a receiver requests retransmission, it fills the fd field with the same value sent from the sender. According to the fd field in the request message, the sender side checks relevant file information. If that file is expired, but the last File ID related with this fd matches the File ID field in the request message, the FILE_TIMEOUT message will be sent to the receiver side. If not, there is no need to send such a message. Similarly, if there is a new file opened with the same fd, which means the new File ID is not identical to the one in the request message, there is also no need to send a FILE_TIMEOUT message, because the new file's BOF or DATA or EOF message will arrive at the receiver side and notify the receiver that the previous file is expired.

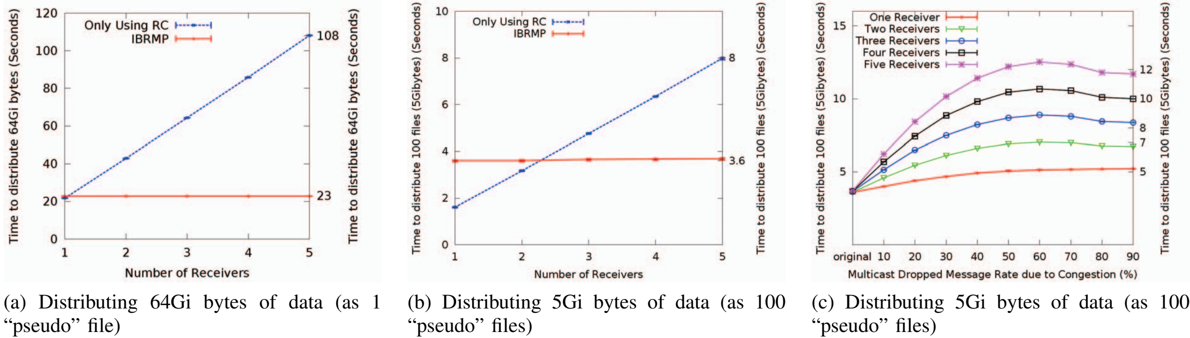
For instance, initially file A is opened and is being sent to receivers, and assume the fd is 3 and File ID is 5. Tuple ⟨fd, File ID, last_File_ID⟩ is used to specify file information. In this case, file A's tuple is ⟨3, 5, x⟩ in which x means it is the first time fd 3 is used. If file A expires, we modify this tuple to ⟨3, -1, 5⟩. If a request message with tuple ⟨3, 5⟩ is received, the sender side will send a FILE_TIMEOUT message to the receiver, but if a request with ⟨3, 4⟩ comes, the sender side ignores such a request message. Then, file B is opened with the same fd 3, and File ID is 6. Now the tuple is ⟨3, 6, 5⟩. If a request with tuple ⟨3, 5⟩ is received, the sender side ignores such a request since any message with File_ID 6 will notify the receiver that the file with File_ID 5 is expired.

IV. Performance Evaluation

Tests were performed on a cluster with six platforms: two equipped with Mellanox[19] Fourteen Data Rate (FDR) CAs; others equipped with Mellanox[19] Quad Data Rate (QDR) CAs. Platforms with FDR MT27500 CAs consist of twin 4-core Intel Xeon 2.4GHz Sandy Bridge CPU and 64GB RAM. Two platforms with Mellanox[19] QDR MT26428 CAs consist of twin 6-core Intel Xeon 2.93GHz Westmere CPU and 64GB RAM. The other two platforms with Mellanox[19] QDR MT26428 CAs consist of twin 6-core Intel 2.93GHz CPU and 6GB RAM. All CAs utilize 2048-byte MTUs, and all platforms are running OFED 3.5-2[18][20] on Scientific Linux 6.3 with kernel version 2.6.32. All platforms are connected via a Mellanox SX6036 switch.

Firstly we compare IBRMP with another data distribution approach which only uses RC to disseminate data among a group of receivers. In this RC version there is one sending agent thread for each receiver. Each sending agent has its own private RC connection and buffer pool, and has no interaction with other sending agents. Each agent tries to keep 32 messages simultaneously in transit. At the start of transmission, each receiver posts 32 receive buffers, and each sending agent posts 32 send messages to its peer receiving side. Each time a receiver receives a message, it sends an explicit ACK back to its sending agent, thereby giving that agent a credit to send a new message. There is only one physical CA on the sender shared by all connections to receivers.

For all our tests we transfer "pseudo" files because data is sent memory-to-memory with no file system involvement. In the first test, we distribute one 64 Gibi (Gi) bytes of data from the sender to groups of one through five receivers. Fig. 5a illustrates this comparison. When there is only one receiver, the approach which uses only RC to distribute data is slightly faster than IBRMP due to the additional retransmission request/response messages and BOF/EOF messages in IBRMP. However, as the number of receivers increases, the approach which uses only RC has to send the same message several times, once for each receiver. As shown by the dashed line in Fig. 5a, the time using only RC to distribute 64Gi bytes of data increases rapidly as the number of receivers increases. For instance, with 5 receivers, the time to transfer 64Gi bytes is 108 seconds because the same data is sent five times. On the other hand, when IBRMP distributes data, most bytes are



(a) Distributing 64Gi bytes of data (as 1 "pseudo" file)

(b) Distributing 5Gi bytes of data (as 100 "pseudo" files)

(c) Distributing 5Gi bytes of data (as 100 "pseudo" files)

Figure. 5: IBRMP performance in transferring large amount of data and various congestion conditions

multicast only once regardless of the number of receivers, as shown by the solid line in Fig. 5a. This scalability is due to InfiniBand multicasting, which duplicates data as needed at adapters and/or switches, thereby reducing communication traffic and minimizing overhead on the sender side. The time to distribute 64Gi bytes of data by using IBRMP is a constant 22.75 seconds, even as the number of receivers increases. Compared to the approach which only uses RC, the improvement with IBRMP is proportional to the number of receivers – with 5 receivers, IBRMP is about 4 times faster.

Fig. 5b compares the transmission time between IBRMP and the RC version during the second test, when 100 "pseudo" files are disseminated from the sender to the same group of receivers simultaneously. Each file is 50 MebiBytes (MiB), for a total of around 5GibiBytes (GiB). In both tests, the RC version has to send the same data several times, which increases the traffic on the sender's CA, which in turn results in inefficient data transmission. IBRMP, by its use of multicast, illustrates its stability and scalability in distributing multiple files.

Normally, multicast datagrams get lost when the receiver is not ready to receive them or due to network congestion. In our testing environment, more than 98% of data are received on the multicast link, which means only a few multicast data messages are lost.

Since the observed loss on the multicast link is small during normal operation, we simulated various network loss environments due to congestion by intentionally dropping multicast messages randomly on the receivers. This forces more messages to be sent back to the sender side to request data retransmission, and consequently more data messages to be sent via RC to each requesting receiver. Therefore, the IBRMP performance is degraded. Fig. 5c shows the total time to transfer the 100 "pseudo" files simultaneously under different network loss environments caused by congestion. The label "original" on the x axis indicates no simulated network loss is injected, which is our original testing environment. As the network loss rate increases, IBRMP takes more time to disseminate data because more data are retransmitted via RC. At a network loss rate of 90% with five receivers, it takes about 11.7 seconds for IBRMP to disseminate these 100 files, compared to the 8 seconds by the approach which only uses RC in our original testing environment (Fig. 5b). Since our test program intentionally drops multicast packets on the receivers, dropped packets are actually received twice. Therefore, the elapsed time in this test (11.7 seconds) is about equal to the sum of the times (11.6 seconds) both approaches take in the original environment.

We also compared our IBRMP with the MPI_Bcast operation implemented in Open MPI[21]. Fig. 6 compares IBRMP with Open MPI 1.6.5 in two tests. In the first test, 64Gi bytes of data are distributed. In the second test, we measured the time that a two byte message is distributed to a group of receivers. Both IBRMP and MPI_Bcast are able to provide low latency service. In addition, IBRMP shows its efficiency and scalability in large amount of data transmission.

V. Conclusion

Unreliable InfiniBand multicast cannot by itself meet the increasing need to distribute data reliably among multiple clusters in data centers and other HPC fields because it is based on unreliable communication, though it can provide efficient and scalable data distribution. In order to bridge the gap between reliable data distribution and unreliable InfiniBand multicast, this paper has described the design, implementation, and performance of IBRMP, a reliable

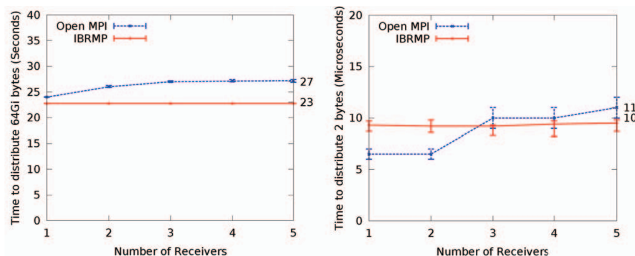


Figure. 6: Comparison between IBRMP and MPI_Bcast

multicast protocol for InfiniBand. IBRMP is provided as a library built on top of OFED, so it can be integrated with OFED. It fully utilizes the advantages of InfiniBand multicast, and guarantees reliable data distribution across an InfiniBand Fabric. In order to minimize run-time resource utilization, IBRMP uses InfiniBand features such as Shared Receive Queues, and gives users control over the number and affinity of sender retransmission threads.

IBRMP uses flow control and ACK batching to prevent possible ACK implosion. It also prevents "ghost receivers" which could delay the notification of transfer completion from IBRMP to a user-level application, because "ghost receivers" generally don't know the transmission status of a current multicast group. Also, IBRMP implements a timeout mechanism to release system resources periodically, and to guarantee robustness for slow receivers. In addition, the selective sending of FILE_TIMEOUT messages guarantees that one is sent only if necessary, reducing overhead and saving resources for possible data retransmission.

Our performance evaluation shows that, compared to an approach which only uses RC, the improvement with IBRMP is proportional to the number of receivers. Because InfiniBand multiplexing duplicates messages at switches instead of at the sender, it reduces communication traffic and overhead at the sender. Therefore, IBRMP benefits from this efficiency. It is able to provide low latency service, and is scalable to a large number of receivers. In the future, we plan to add more receiver nodes and additional switches into our testing environment in order to demonstrate IBRMP performance in a larger, more complex topology.

ACKNOWLEDGMENT

The authors would like to thank our funding agency. This research is supported in part by National Science Foundation grant OCI-1127228.

REFERENCES

[1] Infiniband Trade Association, "Infiniband Architecture Specification Volume 1, Release 1.2.1," Nov. 2007.
[2] Top500 SuperComputers, "http://www.top500.org/."
[3] Internet Data Distribution, "http://www.unidata.ucar.edu/software/idd/."
[4] Local Data Manager, "http://www.unidata.ucar.edu/software/ldm/."
[5] D. Towsley, J. Kurose, and S. Pingali, "A Comparison of Sender-Initiated and Receive-Initiated Reiable Multicast Protocols," in *IEEE Journal on Selected Areas in Communications, Vol 15, No.3*, April, 1997.
[6] B. N. Levine and J. Garcia-Luna-Aceves, "A Comparison of Reiable Multicast Protocols," in *Multimedia Sytems, Vol 6*, 1998.
[7] M. den Burger and T. kielmann, "Collective Receiver-Initiated Multicast for Grid Applications," in *IEEE Transactions on Parallel and Distributed Systems, Vol 22, No.2*, Feb. 2011.
[8] J. Li, M. Veeraraghavan, S. Emmerson, and R. D. Russell, "VCMTP: A Reliable Message Multicast Transport Protocol for Virtual Circuits," in preparation.
[9] J. M. Byers, M. Luby, and M. Mitzenmacher, "A Digital Fountain Approach to Asynchronous Reliable Multicast," in *IEEE Journal On Selected Areas In Communications*, Vol. 20, No. 8, Oct. 2002.
[10] S. Floyd and et al., "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing," in *Proc. ACM SIGCOMM '95*, 1995.
[11] S. Paul, K. K. Sabnani, J. Lin, and S. Bhattacharyya, "Reliable Multicast Transport Protocol (RMTP)," in *IEEE Journal on Selected Areas in Communications, Vol 15, No.3*, Apr. 1997.
[12] B. Rajagopalan, "Reliability and Scaling Issues in Multicast Communication," in *Proceedings of ACM SIGCOMM '92*, Oct. 1992, pp. 188-198.
[13] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Proc. First Workshop Economics of Peer-to-Peer Systems*, June 2003.
[14] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr, "Chainsaw: Eliminating Trees from Overlay Multicast," in *Proc. Fourth Int'l Workshop Peer-to-Peer Systems (IPTPS '05)*, Feb. 2005.
[15] J. Liu, A. R. Mamidala, and D. K. Panda, "Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.
[16] T. Hoefler, C. Siebert, and W. Rehm, "A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, 2007.
[17] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A Remote Direct Memory Access Protocol Specification," RFC 5040, Oct. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5040.txt
[18] OpenFabrics Enterprise Distribution, "High performance server and storage connectivity software for field-proven RDMA and Transport Offload hardware solutions," 2008. [Online]. Available: www.mellanox.com/pdf/products/software/OFED_PB_1.pdf
[19] Mellanox Technologies, "http://www.mellanox.com."
[20] OpenFabrics Alliance, "http://www.openfabrics.org."
[21] Open MPI: Open Source High Performance Computing, "http://http://www.open-mpi.org/."