RapidIO and Multicast: Investigations using a real-time chat

September 2016

Author:
Darshana Padmadas

Supervisor:
Simaolhoda Baymani

CERN Openlab Summer Student Report 2016

15 years CERN openlab

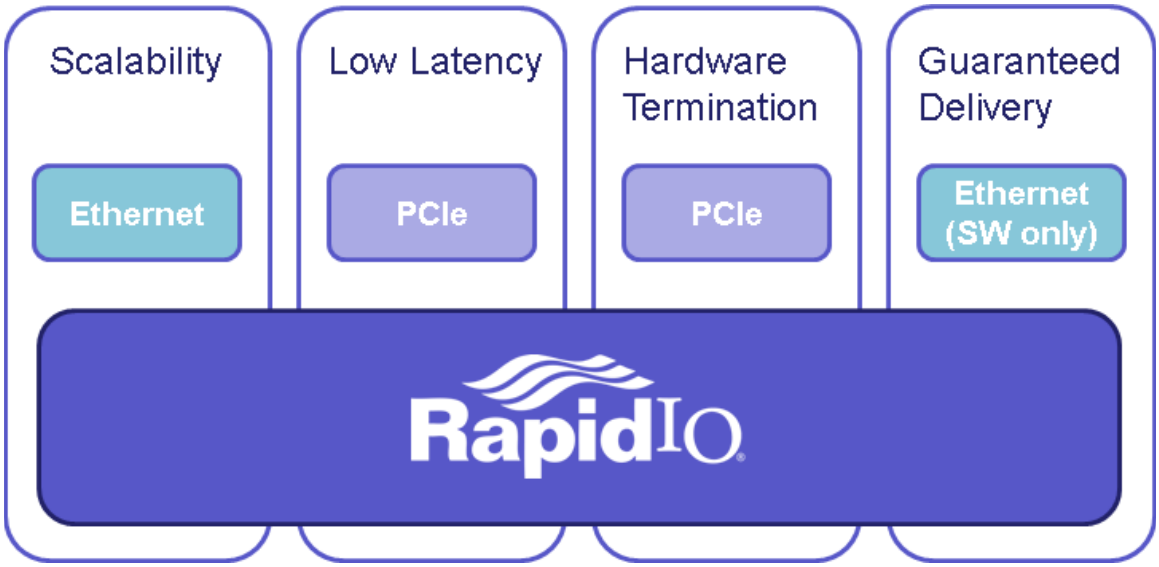
Abstract

RapidIO is a packet-switched high-performance interconnect that is used in 4G/LTE base stations worldwide. It offers several useful features such as low latency, scalability, high availability and guaranteed delivery. This project aims to explore the multicast capabilities of RapidIO technology within a cluster. This is done by creating a chat application which uses RapidIO multicast to distribute chat messages across nodes. Chat messages are sent using multicast remote Direct Memory Access writes (rDMA), which enables one end point to directly write into memory of multiple remote end-points. A locking mechanism was implemented, which ensures only one node performs a multicast rDMA write, so that memory is not over-written. The one-way latency of multicast rDMA writes was then measured, so as to compare them with that of sequential rDMA writes.

Table of Contents

1 Introduction

RapidIO is a packetized point-to-point interconnect technology. It is used in 4G/LTE base stations worldwide. It combines several features offered by Ethernet and PCI Express, as shown in the figure below:



Due to these features, RapidIO is a technology that could be of interest for high-speed communication within a computer cluster. Therefore, this project aims to add to previous investigations into RapidIO, by exploring in detail, the multicast capabilities of RapidIO.

In order to explore the multicast capabilities, a multicast chat application has been designed. This application utilizes the communication paradigms offered by RapidIO, to transmit data among the nodes in a cluster. The project then involved implementing possible methods to measure the performance of the multicast chat application with RapidIO in terms of latency and throughput.

This report aims to provide an overview of the multicast chat application that was created. The structure of the rest of this report is organised as follows. Section 2 describes the previous work done with RapidIO. Section 3 describes the hardware setup and RapidIO functionality. Section 4 gives an overview of the multicast chat application. Section 5 gives an analysis of the project. Section 6 describes the conclusion from the project and Section 7 describes future work that could be implemented to improve the multicast chat application.

2 Previous work

Some RapidIO capabilities have been investigated as part of CERN Openlab's collaboration with IDT. The investigation involved the creation of a file transfer application to validate the throughput of RapidIO and the RapidIO library called *libmport*. The hardware setup used was the same as that mentioned in Section 3.1.

The file transfer application has a single sender and multiple receivers. The receivers are started before the sender and they wait for incoming requests from a sender to connect. Upon connection with the sender, the sender and the receivers exchange important information regarding the file before beginning the file transfer. The connection requests, responses, and the initial communication regarding the file to be transferred are done via channelized messages. Finally, the file is transmitted when the sender performs an rDMA write on the receiver.

The investigation yielded a throughput of around 12 Gigabits per second which is very close to the theoretical maximum of RapidIO (14.5 Gbps) over that particular cluster.

Following this application, the multicast chat application developed as part of the current project conducts a deeper investigation into the multicast capabilities of RapidIO.

3 Background

3.1 Hardware setup

The hardware setup is comprised of 4 computers with the following features:

- Asus Z97 Motherboard (4 cores, 6 MB cache)
- Intel Core i5- 4440 CPU @ 3.1 GHz
- 4 GB DDR3 Memory with speed of 1333 MHz
- 500 GB SATA hard drive
- RapidExpress Bridge card that implements the RapidIO protocol

The four RapidExpress bridge cards in each computer are connected to a RapidExpress Switch Box through which they can exchange RapidIO packets with each other. The switch box contains the IDT CPS-1432 switch chip with 8 ports.

The operating system functioning on each computer is Linux Fedora 20 with a customized version of kernel v3.16.7.

3.2 RapidIO

The following are some of the features offered by RapidIO:

- Low latency
- Guaranteed, in order, packet delivery
- Support for messaging and read/write semantics
- Can be used in systems with fault tolerance/high availability requirements
- Efficient protocol implementation in hardware
- Low system power
- Scales from two to thousands of nodes
- Supports zero-copy data transfer

The RapidIO interconnect is accessed via low level API calls to the *libmport* library. The library is comprised of calls for register access, memory allocation and transmitting data.

RapidIO transmits data by utilizing:

- rDMA: It enables an end-point to directly write into the memory of a remote end-point. It supports zero-copy data transfer. rDMA writes are typically used for large data transactions.

- Channelized Messages: These are typically used for orchestration. They involve the transfer of limited sized buffers between two end points.

4 Application overview

The application has the following characteristics:

- Each node has a connection to every other node. Thus each node can send and receive channelized messages. In this application, channelized messages are used primarily for orchestration.

- Data entered by the users are called *chat messages*. These messages are transmitted to multiple receivers by performing multicast rDMA writes.

- A locking mechanism has been implemented to ensure only one node performs a multicast rDMA write at a time. When a node, has the “lock” it can send its chat messages. This lock is passed among the nodes in a round robin fashion.

4.1 Application workflow

The following describes the workflow in the sender:

1. The sender performs a multicast rDMA write to the receivers;

2. After an rDMA write call, the sender sends a channelized message to each receiver as a notification that an rDMA write has occurred;

3. The sender then awaits for a channelized message from all receivers as an acknowledgment that a receiver has read its respective DMA message;

4. The sender then sends the lock for performing a DMA write to the next node.

The following describes the workflow in the receiver:

1. The receiver waits for a channelized message from the sender that serves as a notification that an rDMA write has occurred. Then it reads the message;

2. The receiver then sends a channelized message acknowledging that it has read the chat message;

3. It then waits for a last channelized message. If the message contains the lock, the node can send its messages.

4.2 Base infrastructure

4.2.1 Establishing connections

The application uses channelized messages for orchestration to ensure the proper order of execution of multicast rDMA writes. For this, all the nodes need to be connected to each other. Therefore each node has *N*-1 connections, where *N* is the number of nodes.

The procedure to establish connections has been adapted from the connection procedure used in DAQPIPE, a small benchmark application designed to test network fabrics for future LHCb upgrades at CERN. The procedure is as follows:

- Node 0 listens for *N*-1 incoming connections.
- Node 1 connects to node 0 as a client, then listens for *N*-2 incoming connections.
- Node 2 connects to node 0 and 1 as a client, then listens for *N*-3 incoming connections.
- Node X connects to node 0-(X-1) as a client, then listens for *N*-X-1 incoming connections.

The following functions implement the connection procedure and have been adapted from the RapidIO driver for DAQPIPE:

- *int OpenServerSocket(struct rio_fd *conn)*:
  Creates a socket that listens for incoming connections from other nodes.

- *int AcceptIncomingConnections(struct rio_fd *conn)*:
  Accepts incoming connections from other nodes.

- *int OpenOutgoingConnections(struct rio_fd *conn)*:
  Sends outgoing connection requests to other nodes.

- *int CloseClientConnection(struct rio_fd *conn)*:

  Closes all connections with other nodes.

4.2.2 Allocating memory for rDMA buffers

The following functions have been adapted from ROOT, a data analysis framework created by CERN, to allocate memory for DMA receive and transmit buffers:

- *int AllocDMARx(char ** rx_buf, int dma_size, riomp_mport_t * mport_hnd, uint64_t * dma_rx_handle, uint64_t * dma_rx_tgt_address)*:

  Allocates and maps an inbound window from the specified target address, for incoming rDMA buffers. For multicast rDMA writes, this target address is the same for all nodes and in this application is set to 0x200000. For unicast rDMA writes, any target address can be used and is typically set to *RIO_MAP_ANY_ADDR*.

- *int AllocDMATx(char ** tx_buf, int kbuf_mode, int32_t dma_size, riomp_mport_t * mport_hnd, uint64_t * dma_handle)*:

  Allocates memory for the outgoing or transmit DMA buffer.

4.2.3 Sending chat messages

The following function is used to send chat messages:

*int SendDMA(struct rio_fd *conn)*

The *SendDMA* function sends chat messages by performing rDMA writes. Based on the destination device ID the function performs unicast or multicast rDMA writes. Before *SendDMA* is called, the destination ID must be configured. In this application, for multicast rDMA writes, the destination device ID is set to 8.

4.2.4 Reading chat messages

The following function is used to read a chat message:

*int RecvDMA(struct rio_fd *conn)*

The *RecvDMA* function reads the chat message from the receive DMA buffer.

4.2.5 Locking Mechanism

Every node allocates an inbound window with the same specified target address. Simultaneous writes within this window will result in scrambled messages. Thus, it must be ensured that at a given time only a single remote node performs an rDMA write. For this, a locking mechanism has been implemented.

This lock is transmitted as a channelized message. When a node has the lock, that is, it receives a channelized message that comprises the string “LOCK”, it can perform a multicast rDMA write of its input messages. After the rDMA write is complete, the lock is passed to the next node in a round robin fashion.

4.2.6 Multicast and sequential rDMA writes

Multicast rDMA writes are implemented using *libmport* library calls. The chat application utilizes RapidIO multicast to perform simultaneous rDMA writes on multiple receivers. To investigate the performance of multicast rDMA writes, they are compared with unicast rDMA writes. In order to replicate the multicast functionality using unicast messages, the chat application performs unicast rDMA writes sequentially, on all receivers.

4.3 Interface

The multicast chat application has a command line interface that is easy to use. The following command is to be executed to start the multicast chat application on a node:

. /multicastchat <destination ID> <number of rounds> <buffer size> <write mode>

destination ID	Destination ID of the node
number of rounds	A single round consists of a multicast rDMA write by all nodes.
buffer size	Size of the DMA receive buffer
write mode	1 for multicast rDMA writes, 0 for sequential rDMA writes

5 Analysis

To evaluate the performance of the multicast chat application with RapidIO, the latency and throughput for multicast rDMA writes were compared with that of sequential rDMA writes.

To compare the latencies and mean speeds for multicast and sequential rDMA writes, a Python script was created. The script runs the multicast chat application on all four nodes, taking as input the number of rounds to be performed, where each round comprises a multicast or sequential rDMA write by every node. The program then collects the time stamps from all nodes, for all rounds, and for different DMA window sizes. With these time stamps, the mean speeds are calculated corresponding to the DMA window sizes for each node. Graphs are then plotted for the mean speeds and the corresponding window sizes for all nodes, comparing the performance of multicast rDMA writes with that of sequential rDMA writes.

5.1 Latency

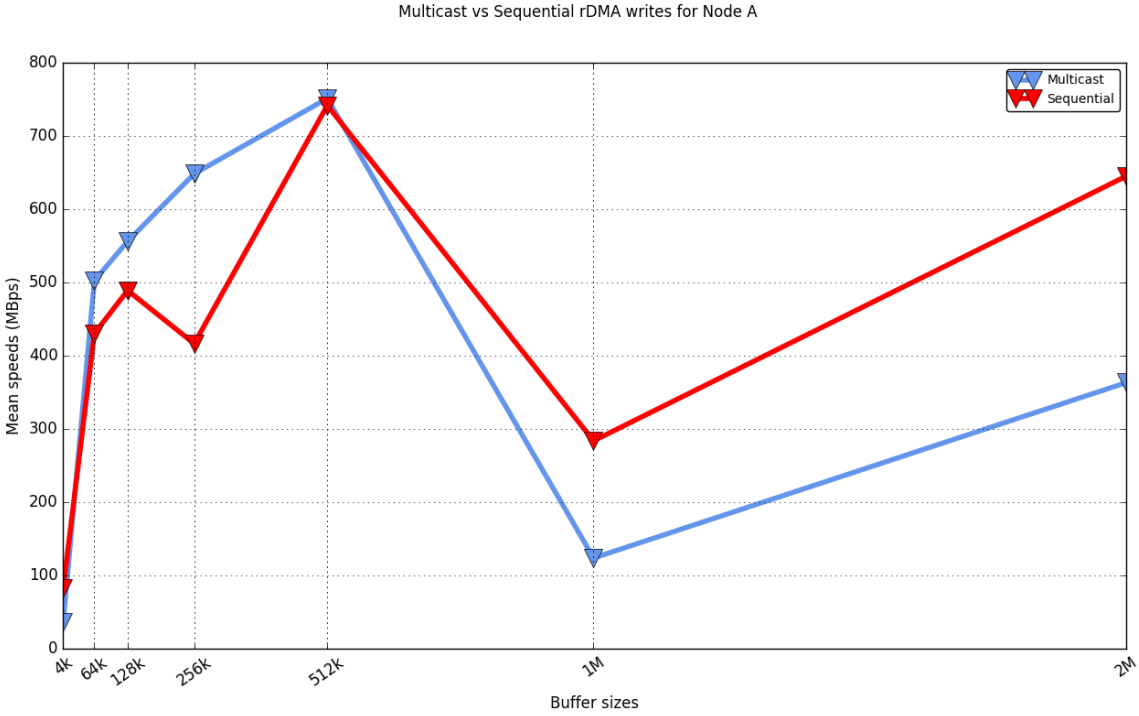
In order to calculate the latency or the time taken for a chat message to be sent and received, the time elapsed for the completion of an rDMA write were measured. Time stamps were printed on the sender and receiver nodes. The *clock_gettime* function was used to retrieve the time from the system wide clock that measures real time.

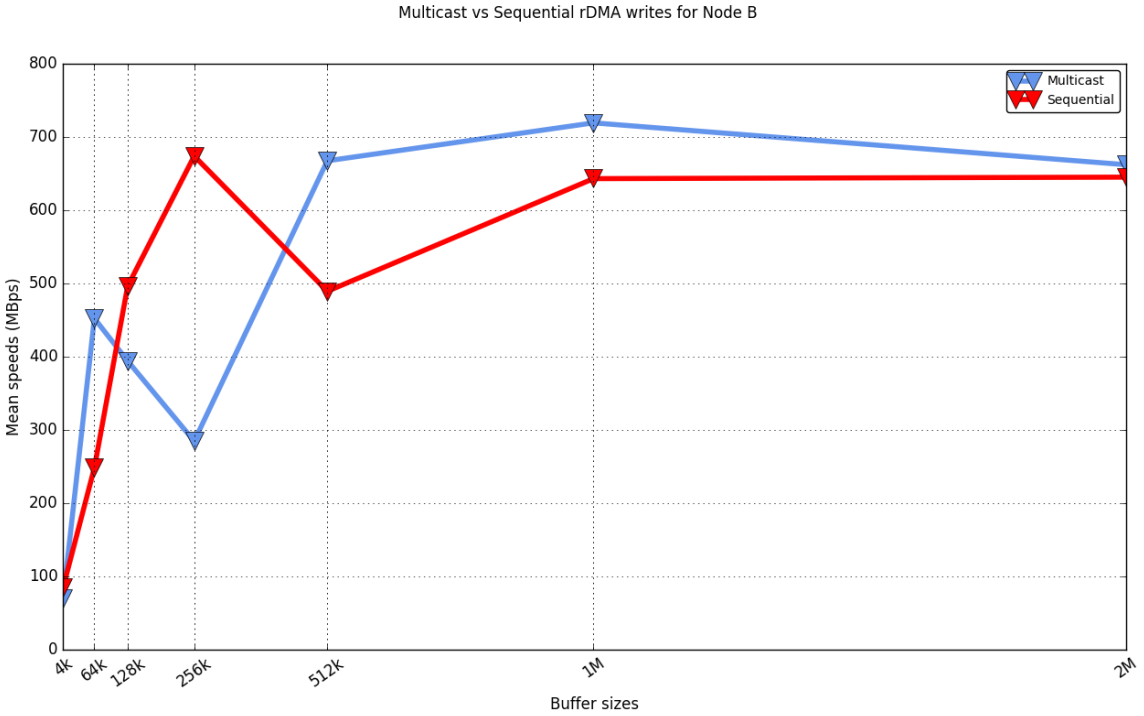
As the hardware setup for the application consists of a cluster with four nodes, each node's clock must be synchronised so as to retrieve time stamps to calculate latencies. The Network Time Protocol (NTP) was used to synchronise the clocks. However NTP only offers accuracy measured in milliseconds, which may not be sufficient for calculating the latencies for the multicast chat application. As a result, sometimes the time stamps retrieved from the receivers have time before that of the sender. With the current setup, the best value for latency that could be obtained was in the order of 10^{-5} seconds.

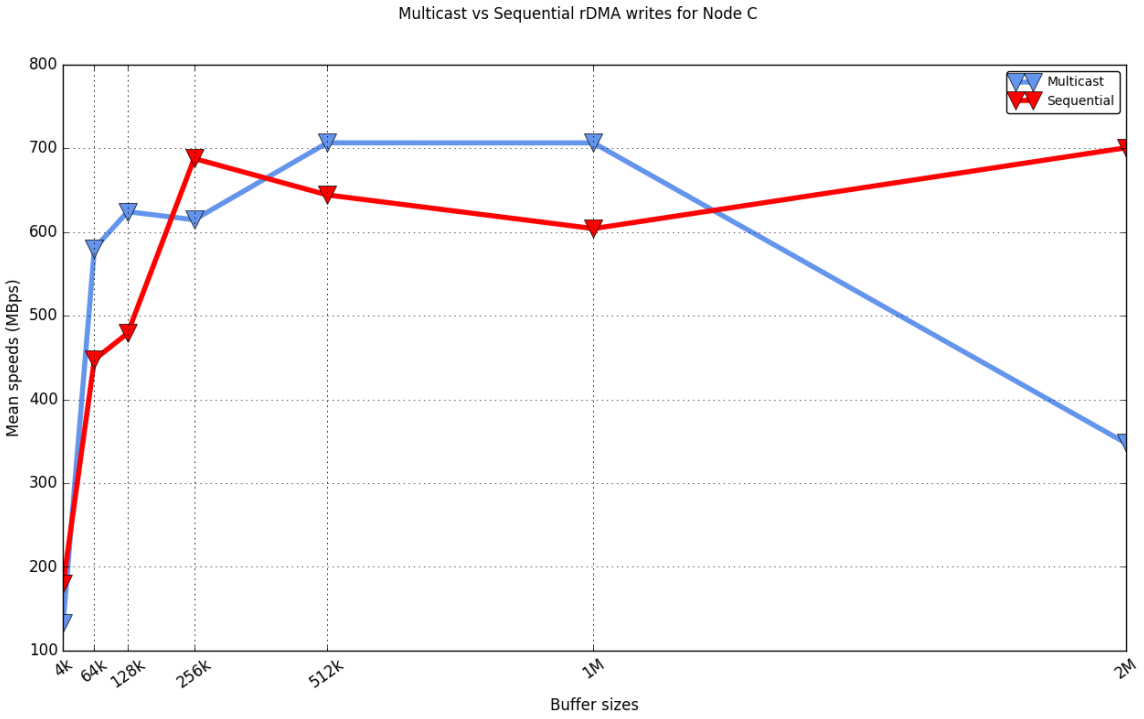
5.2 Throughput

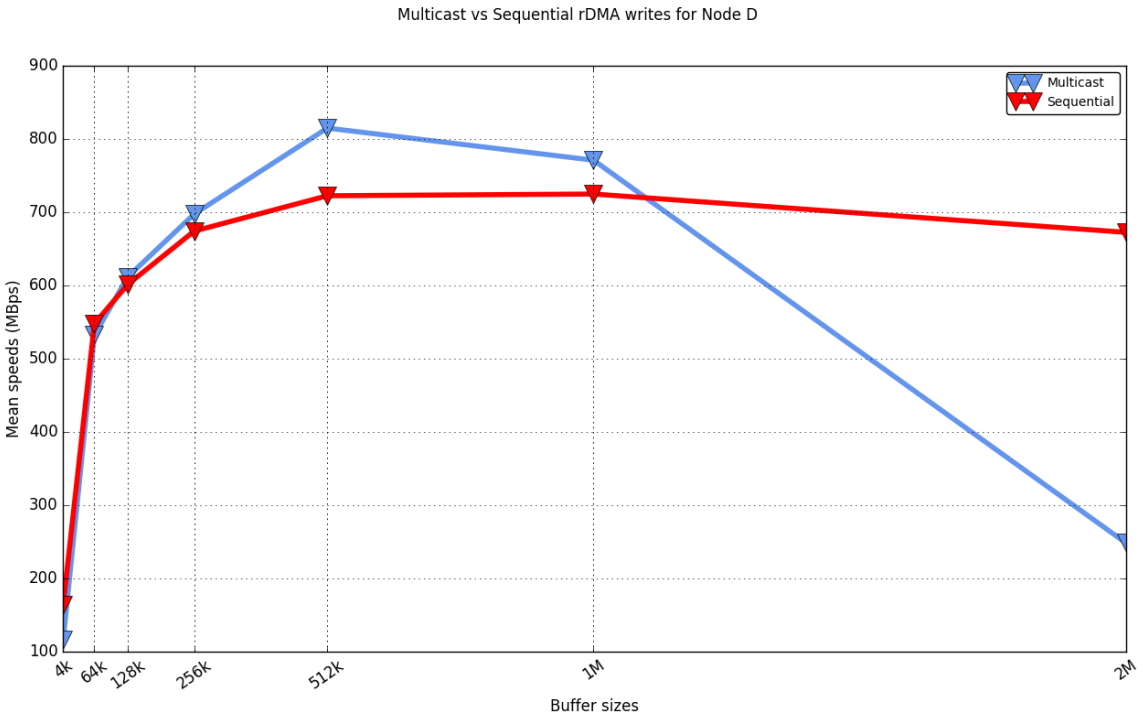
For measuring the throughput of RapidIO for multicast and sequential rDMA writes, different DMA window sizes or buffer sizes were used for the same number of rounds. The mean speed or throughput for every node was calculated by dividing the total number of bytes transmitted from the node during an rDMA write, by the latency of the node. For example, for one round, node A performs a multicast rDMA write to nodes B, C and D with a DMA window size of 4096 bytes, and the latency of A calculated is 10^{-5} seconds, then the throughput is calculated as $(4096*3)/10^{-5}$.

The throughputs or mean speeds were taken and 4 graphs were plotted corresponding to each node. Each graph comprises one plot for the mean speeds for multicast rDMA writes corresponding to varying buffer sizes, and another for the mean speeds for sequential rDMA writes corresponding to varying buffer sizes. The fluctuations in the mean speeds for sequential and multicast rDMA writes could be due to standard error, caching or due to the scheduler. Further investigations into the cause for the fluctuations were not conducted.




Multicast vs Sequential rDMA writes for Node B
Multicast
Sequential
Mean speeds (MBps)
Buffer sizes
0
100
200
300
400
500
600
700
800
4k
64k
128k
256k
512k
1M
2M


Multicast vs Sequential rDMA writes for Node C
Multicast
Sequential
Mean speeds (MBps)
Buffer sizes
100
200
300
400
500
600
700
800
4k
64k
128k
256k
512k
1M
2M



6 Conclusion

This report presents the work that was done in order to explore the multicast capabilities of RapidIO. A chat application was created, that utilizes rDMA writes to send messages. The latencies and mean speeds of multicast and sequential rDMA writes were calculated. The latencies were mostly in the order of 10^{-5} seconds for both sequential and multicast rDMA writes. Graphs that plotted the mean speeds for the corresponding buffer sizes for all nodes were created. As the network protocol used for clock synchronisation within the cluster was not sufficiently accurate, sometimes different time stamps among the nodes were retrieved which yielded negative values for latencies and mean speeds. A more conclusive inference regarding the performance of multicast rDMA writes in comparison to sequential rDMA writes can be made by measuring the latency by calculating the round trip time, as mentioned in the Future work section.

7 Future Work

7.1 Measuring latencies

Currently, one way latency is measured, for which the clocks of all nodes in the cluster are required to be synchronised. Instead of measuring one way latency, a good next step would be to implement functionality to measure the round trip time. Measuring round trip time will yield latencies which include an overhead of the time elapsed for sending/receiving channelized messages.

In order to measure the round trip time, the following procedure can be implemented:

1. The sender allocates an inbound window for receivers to perform a DMA write back. The inbound window is divided into 3 parts for each of the 3 receivers to write back;

2. Before the *SendDMA* function a time stamp is printed. The sender then performs a multicast rDMA write;

3. The sender spawns 3 threads that wait until they receive channelized messages notifying them that the receivers have performed rDMA writes;

4. Upon receiving the channelized messages, the threads exit after printing their respective times;

5. The difference between the time stamps printed by the threads and the time stamp printed before *SendDMA* is calculated, and the largest difference is taken as the latency.

7.2 Creating chat interface

Currently, the DMA transmit message buffer is filled with a rolling alphabet, and the user does not input any chat message to be sent. A new interface can be created that has the following characteristics:

- Contains a prompt, for the user to input a chat message. The user can input messages, which can be sent when the node gets the lock.

- While typing input messages, incoming messages can be displayed. A username/nickname can also be displayed in order to know where the message came from.

- The user prompt should refresh after incoming messages arrive and the input chat message must be displayed again.

In order to handle input from users, and incoming messages from the sender, two threads can be created. One thread can handle the functionality of rDMA writes, synchronisation, and printing messages. The second thread can handle user input.