UiO : Department of Informatics
University of Oslo

Exploring InfiniBand Congestion Control In Mesh Topologies

Mequannint Munye Zeru

Master's Thesis Spring 2016

Exploring InfiniBand Congestion Control In Mesh Topologies

Mequannint Munye Zeru

May 23, 2016

Abstract

InfiniBand is becoming a leading interconnect technology used in many of world's most powerful supercomputers. To guarantee the high-bandwidth and low-latency requirements of HPCs and modern data centers, the InfiniBand architecture supports a congestion control mechanism which specifies a set of parameters. The effectiveness of the mechanism largely depends on these parameter values. InfiniBand is flexible to support several types of topologies which allow simultaneous connections across multiple links. Mesh topology has been one of the most popular interconnection network used in InfiniBand to maintain the bandwidth and latency performance requirements to the highest level.

In this thesis, we studied the effect of the InfiniBand congestion control (IB CC) on the performance of mesh networks and explore a selected set of IB CC parameter values to realize their impact on the behaviour of the mechanism and evaluate the performance gains on this topology. With several number of simulations using CC capable IB model in the OMNeT++ platform, encouraging results that verify the effects of the parameter values and the efficiency of the IB CC mechanism were obtained. The result analysis justified that the negative effects of congestion such as HOL blocking and parking lot problem were successfully avoided by the mechanism and the parameter values *Marking_Rate=1* and *CCTI_Timer=250* improved the network performance by 51.6%.

Dedications

*This work is dedicated to our mother **Yeshi Tefera** who passed away in a situation we can't even attend your funeral.*

"Losing you is still fresh in my memory and it's not getting any easier for me to accept, but I have faith that the God will keep you under his wing and keep you happy for eternity.":- ***Meski***

Acknowledgements

This thesis is the harmonized results of many people. Today, I am highly delighted because I have got the opportunity to express my gratitude for all of them.

- Every work could be successfully accomplished because of the will and interests of the almighty God. Thus, all my first praises and thanks are reserved to him.
- Then, I would like to extend my deepest sense of gratitude to my supervisors **Ernst Gunnar Gran** and **Boning Feng** for their continuous and unreserved support, guidance, constructive comments and suggestions.
- I would also like to take my gratitude to **Oslo University** and **Oslo and Akershus University College of Applied Sciences** for offering this maser programe <nd providing a quality education.
- I would also like to take this opportunity to express my profound gratefulness to **Simula Research Laboratory** for hosting this project and providing suitable research environment.
- My heartfelt thanks is also goes to my lovely wife **Meskerm Kibret** for her continues moral support. *Meski*, you have a special place in my heart and in my life in general. Thanks my sweet.
- My sincere thanks also goes to my friend **Solomon Ayanaw** for his continues support and inspiration. *Sol*, you have been in my life ever since we started our higher education to this day.
- Since this thesis is the cumulative results of the two years learning, I would like to thank all the staff members of the department of *Network and System Administration* involved in the process as well as my classmates for working together in different projects and assignments in harmony.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Interconnection networks have been emerged as an efficient communication infrastructure to realize the heavy communication requirements of the current and future supercomputer systems. Thereafter, it is becoming increasingly pervasive in various application areas such as High Performance Computing (HPC), cloud computing, modern datacenters, and so on. The high performance requirements of these applications also influence the choice of *lossy* or *lossless* interconnection networks. In lossy networks such as the *Internet*, it is possible to drop packets when congestion occurred at some point in the network. However, dropping packets has a consequence of reducing *throughput* and increasing *latency* due to additional time needed for retransmission. This will have a negative impact on the efficiency of the applications in parallel computers and high performance clusters. As a result, the need for very high throughput and low-latency have been the primary factors for the choice of lossless interconnection networks on these application areas. Although the characteristics of these networks are considerably determined by the applications, the performance of most of the communication systems today is highly depends on their topology, routing mechanism, switching strategy and flow control. For instance, networks that implement flow control do not drop packets during regular operation. Therefore, the way that packets are handled by the network drastically constrains the kinds of solutions that can be implemented to address network related problems including, packet routing, congestion, deadlock, and reliability.

In lossless interconnects, in which packets can't be dropped, *congestion* will be one of the most critical and challenging issues that must be dealt and resolved. Congestion is a situation that occurs when bandwidth is insufficient for the network data traffic that exceeds the link capacity. If no countermeasures are taken, congestion in lossless interconnects causes long communication delays, blocking of new connections and degrades network performance. A congestion which is developed at one point in the network link may spread across the entire network unless some congestion control mechanism is implemented. Congestion management also known as Congestion Control (CC) is a countermeasure to reduce the consequences of congestion in interconnected networks. Without suitable

congestion management, network throughput may degrade dramatically when the network or part of it reaches its saturation. To alleviate such problems, several congestion control mechanisms and standards are proposed. InfiniBand Congestion Control is among such congestion countermeasures that is mostly used in a HPC supercomputers or modern data centers.

InfiniBand (IB) [20] is a widely used, low-latency, high-bandwidth interconnect standard which describes a congestion control mechanism for detecting and resolving congestion. Since scalability and industry-wide versatility are among the basic characteristics of InfiniBand, many design choices such as topology, routing and congestion control parameter configurations are left intentionally unspecified by the InfiniBand architecture in order to accommodate a wide variety of applications. As a result, InfiniBand congestion control mechanism is rich in the way that it specifies a set of parameters that can be tuned in order to achieve effective congestion control for a variety of network topologies. Appropriately setting these parameters should enable the network to resolve congestion while still utilizing the network resources. These parameters determine characteristics like when switches detect congestion, at what rate the switches will notify destination nodes, and how much and for how long a source node contributing to congestion will reduce its injection rate. Furthermore, these set of parameters add flexibility to the InfiniBand congestion control mechanism.

1.1 Problem Statement

InfiniBand Congestion Control (IB CC) provides functions such as congestion detection, congestion signaling, injection rate reduction and injection rate recovery for high performance computers and data centers. InfiniBand supports congestion control with a parametrized mechanism where the value of the parameters has significant effect on the performance of the network. However, one of the major challenge with the congestion control mechanism found in InfiniBand is how to configure these parameters properly for a given system. Exploring the proper parameter space in a large scale cluster and properly configuring IB CC in various topologies of different sizes is a very difficult and time consuming task.

The InfiniBand congestion control specification doesn't provide proper guidance on how to configure IB CC parameter values for the congestion control mechanism to be efficient. As a result, it is usually very difficult to configure and activate a congestion control mechanism in production clusters and supercomputers due to concerns that it may negatively impact the network performance if the mechanism is not appropriately configured. This is because the behavior of the network heavily depends upon the values of these parameters.

In order to achieve the desired benefit of IB CC, such as improved network resource utilization and hence improved network performance, it is important to study the effect of the congestion control mechanism

for a given application. Finding and setting the appropriate congestion control parameter values for a given topology in a stable and efficient manner requires exhaustive experimentation. Hence, adequate knowledge is required on how to use the parameters, what effect the individual parameters have on network performance, and how they relate to each other. Therefore, the purpose of this thesis is to study and explore a selected set of IB CC parameters in order to understand their impact on the behaviour of the IB CC mechanism and evaluate the performance improvements on various scenarios for mesh topology [6], which will be explained later in Section 2.1.1. The thesis attempts to address the following research questions:

- How do the proper IB CC parameter values are set to ensure the efficient behavior of the IB CC mechanism for mesh topologies?

1.2 Research Methods

There are three fundamental methods for network performance evaluation and analysis: *empirical measurement*, *analytical modeling* and *simulation* [3, 40]. Empirical measurement is a research method which is based on observed and measured phenomena. The measurements or data are collected through direct observation or experimentation. Empirical measurement method is possible only if something similar to the proposed system that allows to collect the specific performance information already exists. Empirical measurement is the most accurate of the methods but it is an expensive research method since it requires a system to be implemented. In situations in which measurement is not possible, analytical modeling and simulation can be used. In this thesis, the system size required will be very large, unfortunately, we do not have access to such large, very expensive systems. Therefore, measurement techniques won't be the appropriate method for this thesis.

Analytical modeling is a set of equations describing the performance of a system. It includes techniques such as queueing networks, Markov models, state charts, etc. Analytical modeling maps the subject of the research to a mathematical or statistical model. Then it describes a collection of measured and calculated behaviors of the network elements over a finite period of time. It is the least expensive method since it doesn't need hardware and software to be implemented. However, the complexity of the model grows radically when the network is advanced and includes more details unless some assumptions made to simplify the network model. Since the network we considered in this thesis needs all relevant details related to congestion management to be handled, analytical methods couldn't be a good choice.

Simulation is the abstraction of the functions of a real system using a software model. Simulations are inexpensive tools for testing, verifying and generally experimenting with new technologies/features in a repeatable fashion. It is a useful and widely used technique for network performance analysis, but are often time consuming and difficult to conduct. The

goal of every performance analysis is either to compare different alternatives. Simulation provides an easy way to predict the network performance or compare several alternatives and often produces a fairly accurate performance measurement results. Performance analysis which is based on simulation requires designing a proper set of experiments for simulation. In our work, simulation will be the research method since it is an inexpensive method that will allow us to conduct a set of large-scale and multiple simulation experiments to explore the parameter values. A congestion control capable InfiniBand model [13] simulator implemented in the OMNeT++ platform [32] will be used for simulation.

1.3 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides an overview about the basic building blocks of interconnection networks such as topology, routing, switching strategy and flow control techniques. Furthermore, it provides a brief description on congestion in lossless interconnects, congestion control and its parameters in InfiniBand networks and the OMNeT++ network simulation platform. Chapter 3 describes the simulation setup, the planned methodologies to address the problem statement, the experiment design and simulation result collection, processing, plotting and analysis strategies. Chapter 4 presents and analysis the results obtained through simulations. Chapter 5 provides further critical analysis and discussions on the outcomes of the thesis. Chapter 6 summarizes the thesis along with major contributions and concludes with possible future research directions.

Chapter 2

Background

This chapter presents a brief introduction on the basic concepts, terminologies, techniques, and technologies used in this thesis. It starts with the basic building blocks of interconnection networks and continues details on congestion and congestion control mechanisms on lossless interconnects such as InfiniBand. Finally, it deals with OMNeT++ simulation environments and the simulation model used in this thesis.

2.1 Interconnection Networks

Interconnection networks cover a wide variety of domains ranging from On-chip networks in the internal buses of computer circuits to a wide area computer networks which connect computer systems that are distributed across the globe [8]. However, the focus of this thesis is on large scale high-performance point-to-point interconnection networks established between compute resources, storage and switches to function as a communication backbone of supercomputers or HPC clusters. Interconnection network consists of two basic elements: a set of processing elements called *nodes* and a set of communication media called *channels*. A channel is a physical link between network nodes. Network nodes which include host channel adapters (HCAs), switches and routers, have the functions of generating, routing, forwarding, and consuming network traffics. HCAs are usually connected to a single switch or router and responsible to generate or consume a network traffic that can be transferred from a source host node to a destination host node along a path, or *route*, which comprised of a sequence of channels and switches or routers. Interconnection networks [6] can be characterized by its topology, routing algorithm, switching strategy, and flow control mechanism. In the following sections we will provide introductory explanations on the key concepts and ideas on interconnection network technologies and techniques. The explanation will focus on lossless interconnection networks like InfiniBand networks in HPCs and modern data centers which is the core of this thesis.

2.1.1 Network Topologies

Network topologies describe the physical and logical arrangement of nodes and channels in an interconnection network [6]. The physical topology of a network refers to the configuration of cables, computers, and other peripheral devices, while logical topology shows flow of data within a network, regardless of its physical structure and position [8]. The topology of interconnection networks plays an important role in the performance of all types networking applications [52]. In high performance computing, each node within the cluster needs to be able to communicate with other resources and with other nodes in the cluster for control and inter-process communications. As a result, building a large network topology that support the most demanding HPC environments such as InfiniBand requires careful considerations of the traffic flow.

Currently, there are a range of different types of topologies in use in HPC supercomputers and high performance data centers. The most popular interconnection network topologies used today in high performance computing are Fat-tree, Mesh, Torus and Hypercube [30]. Network topologies can be classified into two major categories [1]: *direct* and *indirect*. Direct topology includes Mesh and Torus topologies in which every node in the network is a source and a destination of traffic as well as forwards traffic from other nodes in the network. Indirect topology includes fat-tree and butterfly topologies in which there is a clear distinction between an end node and a switch. In indirect topology, only end nodes are sources and destinations of traffic, switches simply forward traffic to and from end nodes. InfiniBand is a flexible communication standard that provide support for several types of topologies which include fat-tree, 2D/3D mesh and 2D/3D torus [18]. These topologies allow simultaneous connections across multiple links and provide scalability to connect more nodes and links in the InfiniBand fabric. This makes InfiniBand the today's technology of choice for parallel computing applications that involve a high degree of message passing between nodes. According to the TOP500 LIST results reported on November 2015 [42, 20], 47% of the TOP500 list ranked InfiniBand above Ethernet and other proprietary solutions.

Fat Tree

Fat-tree topologies are widely employed in cluster supercomputers usually using InfiniBand network technology [14, 34]. The fat-tree is an indirect interconnection network which provides multiple paths between individual source and destination pairs which makes it easier to handle network faults. Compute nodes of a fat-tree are located at the leaves of the tree and one or more levels of switches are used to connect the nodes. The links get thicker towards the top of the tree. According to [26], fat-tree topology enables the best performance at a large scale and effectively utilizes computing resources. Since the capacities of a fat-tree increase as we go up the tree, one need not worry about the exact capacities of channels. The InfiniBand congestion control in fat-tree topology can be configured to be stable

for a given installation [14].

A variant of fat-tree known as the *k-ary n-trees* includes a parametric family of regular topologies that can be built varying the two parameters k and n [34], where k defines the number of "downward ports" at each switch and n denotes the number of levels in the tree. Interconnection networks based on the *k*-ary *n*-tree topology are widely used in high performance computing that use InfiniBand networks. Figure 2.1 shows an example of 4-ary 2-tree fat-tree which has 8 switches and 16 nodes.

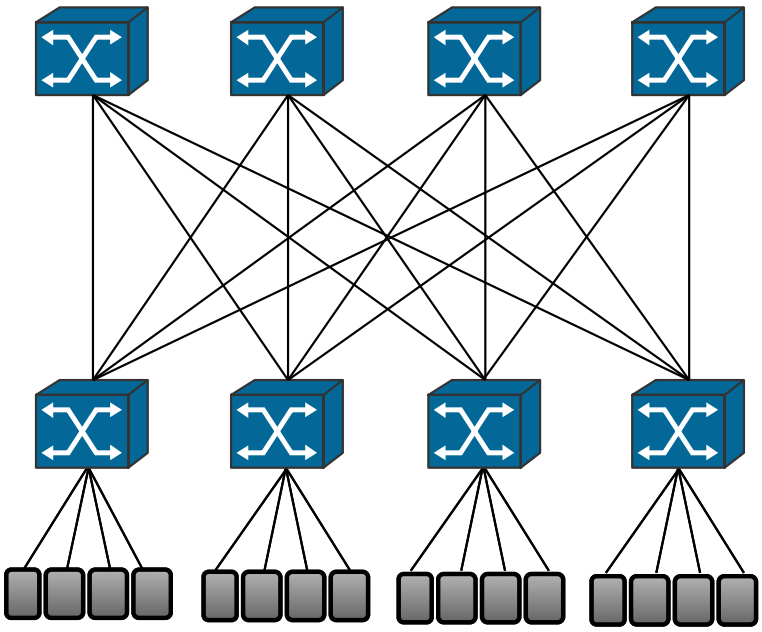

Figure 2.1: 4-ary 2-tree Fat-Tree.

2D/3D Mesh

A mesh topology is a direct network topology that form a grid structure, usually in two or three dimensions. The 2D mesh network topology is a 2D grid of nodes where each node is connected to its own switch. A 3D Mesh is a logical extension of 2D by adding one more dimension. Torus and mesh networks are a family of *k*-ary *n*-cubes, where $N = k^n$ number of nodes in a regular *n*-dimensional grid with *k* nodes in each dimension and channels between nearest neighbors [6]. Figures 2.2 and 2.3 show a 2D (4-ary 2-cube) and a 3D (3-ary 3-cube) mesh network topologies of switches, respectively. Mesh topologies can be categorized into *full-mesh* and *partial-mesh* topologies. In a partial-mesh topology, at least one device maintains multiple connections to other without being fully connected. The full-mesh topology connects all devices to each other for redundancy and fault tolerance. In fully connected topology, even though a link can be saturated if a source node is sending more traffic than a destination node can handle, no congestion spreading is possible, as a result, no congestion management is needed. A fully connected topology does not scale, so, there is little

interest in the context of HPC supercomputers or large data centers.

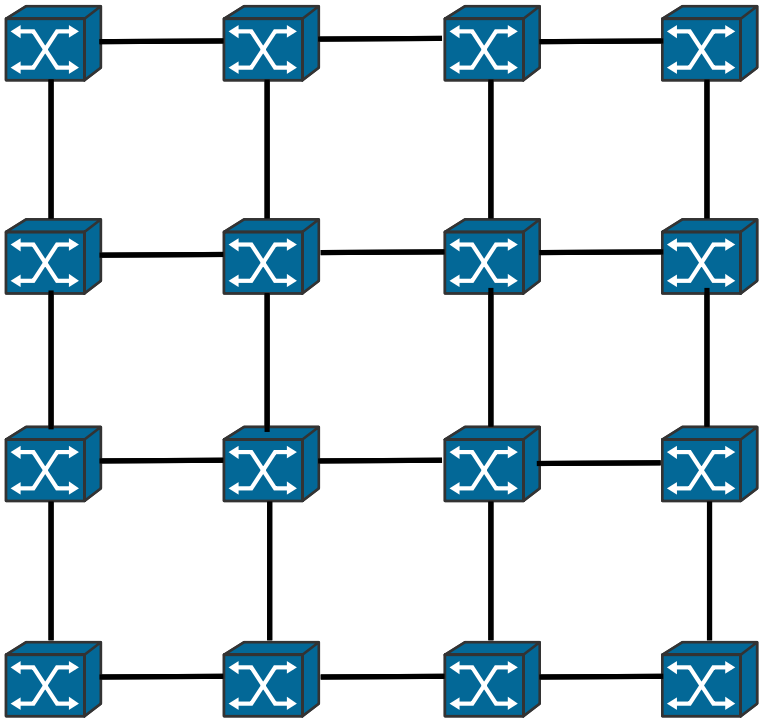

Figure 2.2: 2D Mesh Topology.

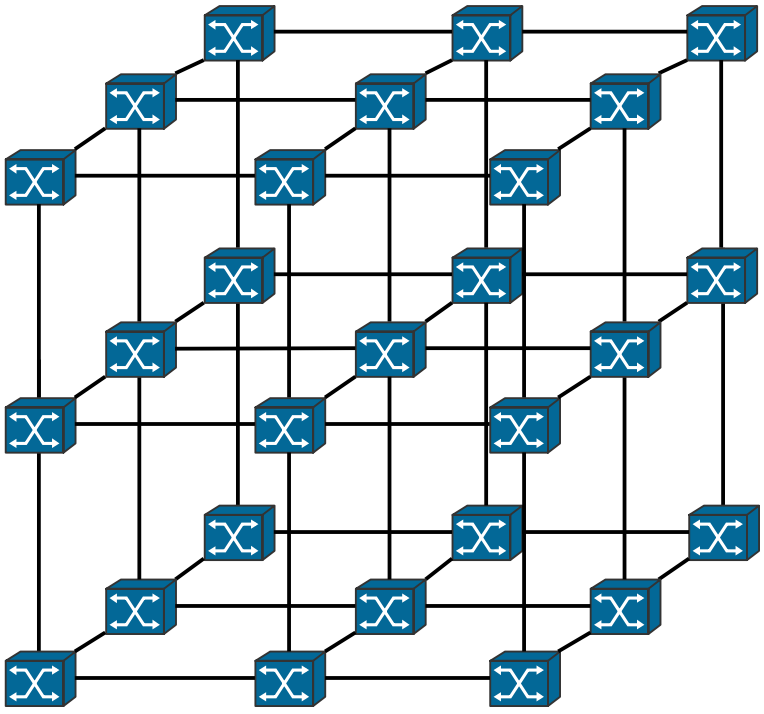

Figure 2.3: 3D Mesh Topology.

2D/3D Torus

Torus is a direct network topology for connecting processing nodes in a parallel computer system. A Torus network is same as a mesh network with boundary nodes connected by wrap-around edges. Which means that torus topology is an extension of a mesh topology in which the end nodes of a mesh topology are wrapped around and connected to the first node. These wrap around edges significantly reduce the overall diameter of the network and thus improving its throughput and latency. As a result, torus is considered as the most efficient interconnection network for current and future supercomputer systems [52, 51]. Torus interconnection networks are often used on high performing supercomputers. Torus and mesh topologies are widely used in high performance computing because these networks allow high-speed operation without repeaters in short distances and can exploit physical locality between communicating nodes with low latency and high throughput. As shown in Figure 2.4 and 2.5, the 2D (4-ary 2-cube) and 3D (3-ary 3-cube) torus network topologies are extensions of 2D and 3D mesh topologies by the inclusion of edges between the exterior nodes in each row and those in each column, respectively.

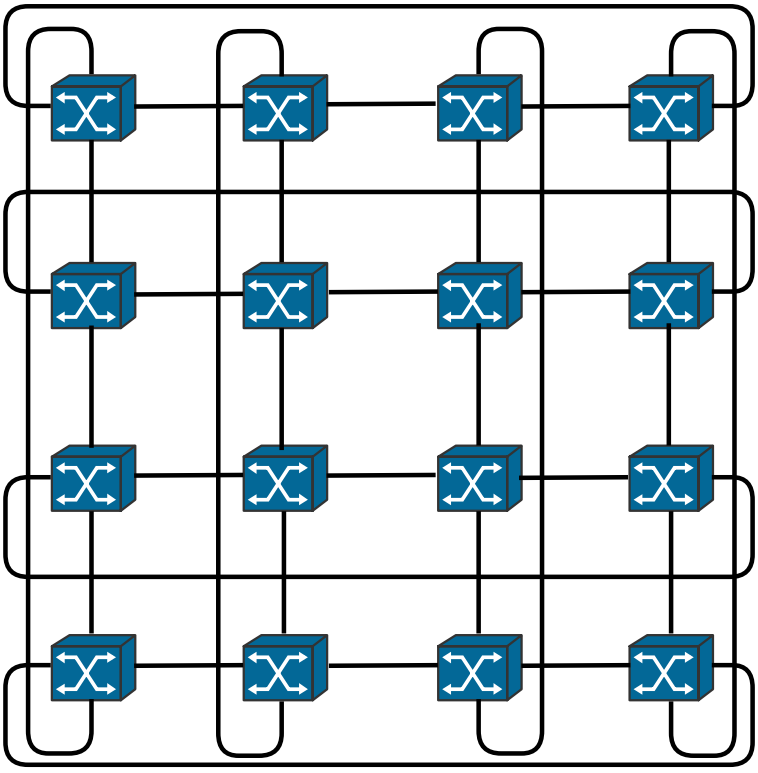

Figure 2.4: 2D Torus Topology.

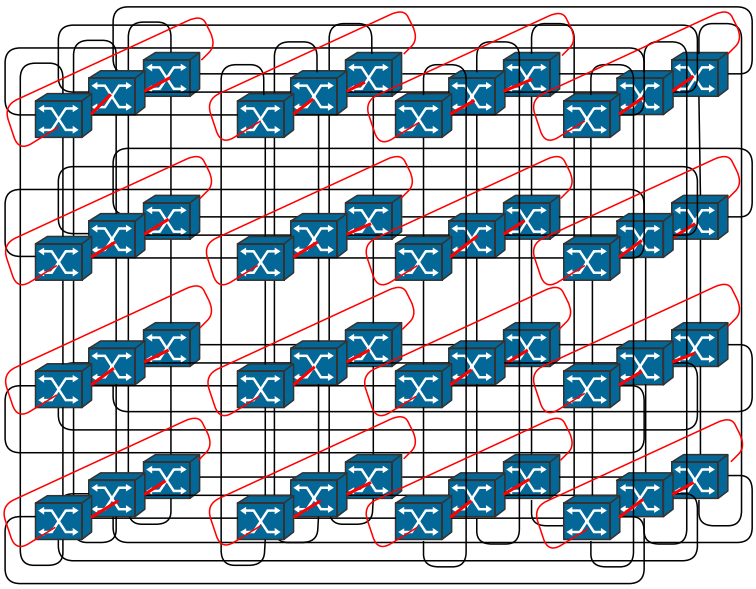

Figure 2.5: 3D Torus Topology.

2.1.2 Routing

While the network topology determines the physical interconnection structure of the network, the routing algorithm determines the path taken by a packet from a source node to a destination node. Routing is the process which involves selecting a path and moving a packet from a source node to a destination node in a particular topology. A good routing algorithm must prevent deadlock [5], livelock[2] and starvation [41], and attempts to balance the traffic load across multiple links. Routing algorithms can be classified from different point of views such as: where the routing decisions are taken(source, centralised and distributed routing), how paths are selected or adaptivity, routing decisions, progressiveness, implementation, number of destinations and so on. Based on the way how they select a set of possible paths from source node to destination node, routing algorithms can be classified as: *Oblivious*, *Deterministic* and *Adaptive* routing algorithms [8, 6].

Deterministic routing algorithms completely specify the path from the source node to the destination node and always choose the same path between every pair of nodes even if there are multiple possible paths. They are the simplest and inexpensive routing algorithms to implement and to easily avoid deadlocks. Deterministic routing algorithms guarantee in-order delivery of packets, i. e. the packets reach to a destination node in the same order they are delivered from a source node. Many networks with irregular topologies use deterministic routing since adaptive routing algorithms are more difficult to implement in such topologies. However, deterministic routing algorithms don't consider path diversity and make effective use of network links of the underlying topology while establishing a path since it always uses a unique path for each source and destination pairs. Moreover, these algorithms don't provide load balancing across

multiple paths because it do not consider the current state of the network.

Oblivious routing algorithms randomly choose one of the many existing shortest paths between the source and destination without taking into account any other information except the addresses. In oblivious routing, the routing decision is made independent of the status of network traffic and at the same time irrespective of how much traffic any other pair chooses to send. They choose the route for a packet from every sender to receiver pair without the knowledge of the actual traffic demands that arise in the network. Deterministic routing algorithms always output the same path between a given source and destination pair. However, oblivious routing may produce different paths at different times for a given source and destination pair although the routing decision is oblivious to the present state of the network. These algorithms are simple to implement and simple to analyze when compared to adaptive routing algorithms.

Adaptive routing algorithms use the present state information of the network while making routing decisions. Instead of picking a path independent of the network state as oblivious routing algorithm, adaptive routing adjust its routing decision based on the current traffic conditions. Adaptive routing algorithms use information such as the status of a node or link, the length of queues on network nodes, and the historical information about the load of the channel in order to avoid congested or faulty regions of the network. However, adaptive routing adds extra complexity and cannot guarantee in-order packet delivery. In a situations where adaptive routing is too expensive and very difficult to implement, oblivious non-deterministic algorithms usually used. Adaptive routing algorithms are also highly coupled with the flow control mechanism since both concerned about the current state of the network to make decisions.

InfiniBand uses deterministic routing [27] which is based on the forwarding table stored in each switch. The forwarding table in each switch maps the local identifier (LID) of an HCA to an output port of the switch. A packet arrived at a switch with the packet destination local identifier (DLID) will be forwarded to the corresponding output port via the forwarding table lookup. Since one LID only mapped to one output port, routing in InfiniBand is deterministic. For n-dimensional k-ary n-cube network topologies such as mesh, torus and hypercubes, which are mostly used in InfiniBand networks, the simplest progressive routing algorithm called *DOR* (Dimension Order Routing) is mostly used. DOR [19] routes the packet first in the x dimension, next in the y dimension, and then in the z dimension so as to determine the shortest path. It is one of the popular deterministic routing algorithms supported in InfiniBand.

2.1.3 Switching Strategy and Flow Control

In interconnection networks *channels* and *buffers* are the basic resource that have to be properly managed. Providing fair and deadlock-free management of these resource is the task of the switching strategy and flow control. The switching Strategy and flow control mechanisms together determine how and when the information is routed along its path. The

switching strategy defines how the data in a message traverses its route while the flow control mechanism defines when the message, or portion of it, moves along its route.

Switching Strategy

Switching is an approach of delivering packets across the network. There are two typical switching strategies available for digital traffic to determine how messages are forwarded through a switch or a router. These are: *circuit switching* and *packet switching*. In circuit switching strategy [23], when a node wants to communicate with another node, a fixed-bandwidth channel called a *circuit* will be created between the source and the destination. This circuit is reserved exclusively for a particular information flow, and no other flow can use it. This path from the source to the destination is established and reserved until the data is transferred over the circuit. Consequently, a dedicated path/link between the source node and the destination node is set up for the duration of communication session to transfer data.

Circuit switching was originally proposed and used in telephone networks. When it is applied to data communication networks, it suffers from some limitations. The first one is the slow path set up which delays transfer of messages from sender to receiver. The other one is, the degradation of the network performance due to the poor utilization of network resources. The efficiency of interconnection networks comes from the fact that communication resources such as channels are being shared. However, when a dedicated channel between two communicating nodes is created and used by only two communication nodes, the channel will be idle for a high proportion of the expensive connection time while others are waiting for the channel to be free. In circuit switching, the routing decision is made when the path is set up across the given network, but there are no decisions made after that time. Circuit switching doesn't require congestion management since resources are reserved in advance to prevent congestion during data transfer.

Packet switching can be defined as the routing of data in discrete quantities called *packets*, each of a controlled format and with a maximum size. In packet switching, the entire message is split up into a sequence of packets of a fixed maximum size to be sent across the network without prior reservation of resources. Each packet contains routing and sequencing information as well as data. Once a message has been transmitted, the packets which comprise it, may take different routes through the communications system and arrive at the receiver at different times. Which means packets are individually routed from the source to the destination and reassembled at the destination node into the correct order to obtain the complete message. In packet switched networks, a congestion control is typically used since no resource reservation is made prior to data transfer. Congestion control is needed when buffers in packet switches overflow or congest. The InfiniBand network is a packet switching based network. Therefore, in this thesis, we will consider packet switching

technique.

Flow Control

The routing of a message in an interconnected networks requires the allocation of various resources such as: channel bandwidth and buffer spaces. Flow control is a mechanism of allocating these resources to data traffics traversing the network. Since the network switch fabric is not self-throttling, each node continues to send a steady stream of packets regardless of the congestion in the fabric. Thus, a flow control mechanism across multiple nodes is needed when a given node is injecting data faster than a network switch or an end node can process it. Flow control is a mechanism for throttling the rate of data exchange between any two points in order to prevent overload and its consequence of packet loss due to congestion. The flow control mechanism ensures the availability of sufficient buffer spaces at the receiver to avoid the loss of data. Consequently, networks that implement flow control do not drop packets due to congestion. Flow control provides a management alternative to having large buffers. However, the usual *back-pressure* effect of the flow control substantially reduces the overall throughput of the network if congestion spreading is not properly managed.

There are several interesting flow control mechanisms including: Xon/Xoff, credit-based, sliding window, and flit reservation which are commonly used to implement flow control. Credit-based flow control [25] is among these popular flow control techniques used in InfiniBand. In this flow control technique, the upstream node needs to receive credits before sending a packet to the downstream node. At various times, the downstream node sends a credit to the upstream node indicating the availability of buffer space and for the maximum number of segments it is willing to receive without loss of packets. Upon receiving credits, the upstream node is eligible to forward some number of credits and may send credits according to the received credit information. Credit-based flow control typically uses a credit counter at the upstream node that initially contains a number of credits equal to the number of buffers at the downstream node. Every time a packet is transmitted, the upstream node decrements the credit counter. When the downstream consumes a packet from its buffer, it returns a credit to the upstream node in the form of a control packet that notifies it to increment its counter upon receipt of the credit. These techniques essentially control the flow of packets into the network by throttling packet injection at the upstream node when the downstream node runs out of buffer space or when the upstream node runs out of credits. In credit-based flow control, buffer overflow will never happen since the upstream node will run out of credits and stops transmission while the downstream node is run out of buffer space.

2.2 Congestion in Lossless Networks

Network congestion is the outcome of high amount of traffic flow that exceeds the network link capacity. Congestion occurs when traffic source nodes inject more data than the network devices such as routers and switches, can consume. The link level flow control mechanism of lossless networks prevents the network from dropping packets as we discussed in Section 2.1.3. Because of this mechanism a congested network can experience an undesired effect. The effect will cause the buffers on such devices to fill up due to congestion. When the situation is intensified, the link level flow control blocks upstream traffic, which in turn may cause buffers in upstream switches to fill up as well. In this way, a congestion that is initially concentrated at a single hot spot can spread throughout the entire network. This phenomenon which is caused by the back-pressure effect of the traffic flow control is called *congestion spreading* and leads to *congestion trees* to be grown up. Congestion spreading starts at a switch, and then blocking can spread further upstream and eventually fill buffers all the way back to the traffic source nodes. This reduces throughput and increases latency on all switches in which buffers fill up.

To demonstrate the situation, lets consider a simple network topology shown in Figure 2.6. This small 2X2 2D Mesh topology consists of the three basic building blocks of interconnection networks. These are:

- *End Nodes*: These nodes may be a source(injector) and a destination(sink) of the traffic flow. In this topology, we have seven end nodes from H1-H7.

- *Forwarding Nodes*: These nodes are usually switches. We have four switches from SW1-SW4.

- *Links*: These are links for bidirectional communication channel between adjacent nodes in the network. In this example, we assumed that the capacity of the switch to switch links is twice the capacity of the switch to node links.

In Figure 2.6 below, three traffic flows, which are indicated with green line, are injected into the network from source nodes H2, H4 and H5. All the three source nodes send the traffic in the full link capacity and all want to send for the same destination node H7. H2 sends its traffic through *H2-SW1-SW3-SW4-H7*, H4 sends its traffic through *H4-SW2-SW4-H7*, and H5 sends its traffic through *H5-SW3-SW4-H7*. These traffic flows shared the network link SW4-H7, which are beyond its capacity. Thus, the link SW4-H7 will be the bottleneck link and the source nodes have to use only their share of the link. Therefore, traffic starts to pile up at SW4. When SW4 is running out of buffer space, the control flow will notify the neighbouring nodes, SW2 and SW3, that SW4's buffer space is fill up and prevents them from forwarding traffic in their full link capacity to SW4. The back-pressure effect of the flow control leads the buffers at SW2 and SW3 to fill up. Furthermore, SW1 will also be prevented from forwarding traffics from H2

to SW3. The three nodes, H2, H4 and H5, which are injecting traffics will be the *contributors* of the congestion. As shown in Figure 2.7, a congestion tree whose root is at SW4, next to the bottleneck link, and its branches towards the contributors of the congestion, which is indicated by red color, will be grown up.

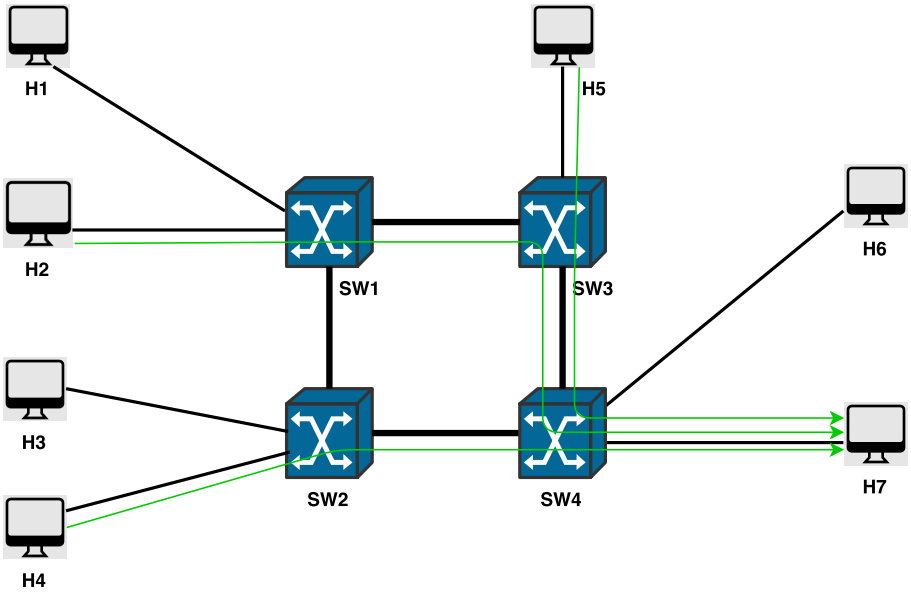

Figure 2.6: Three Traffic Flows.

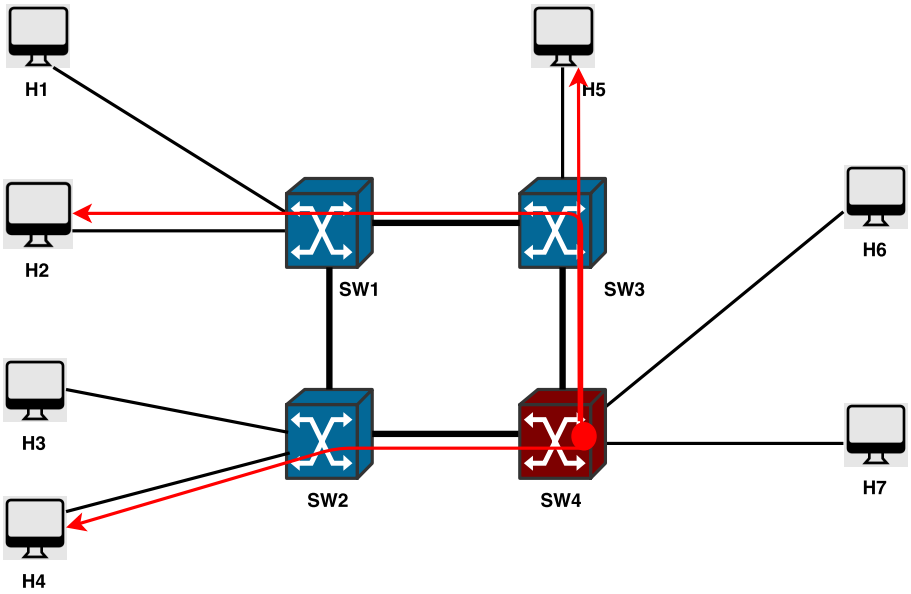

Figure 2.7: Congestion Tree for Contributors.

Since there is no alternative path for the traffic to reach H7, it is mandatory for the source and forwarding nodes to wait for their fair share of the bottleneck link. So, the congestion tree doesn't have a potential harm

on the network performance yet. However, the situation worsens when a new traffic flow which doesn't request the bottleneck link is blocked by the congestion tree. In Figure 2.8, additional traffic, which is indicated with red line, from source node H3 is added to the topology towards the destination node H6. This flow doesn't request the bottleneck link SW4-H7. So, the flow is expected to progress towards its destination without affected by the contributors of the congestion. However, due to the congestion tree which occupied the buffer spaces at SW2 and SW4, H3 should compete for its share of the buffer space. Even though SW2 is a fair switch which alternatively forwards packets from H3 and H4, their share at SW4 will be constrained by only the share of SW2.

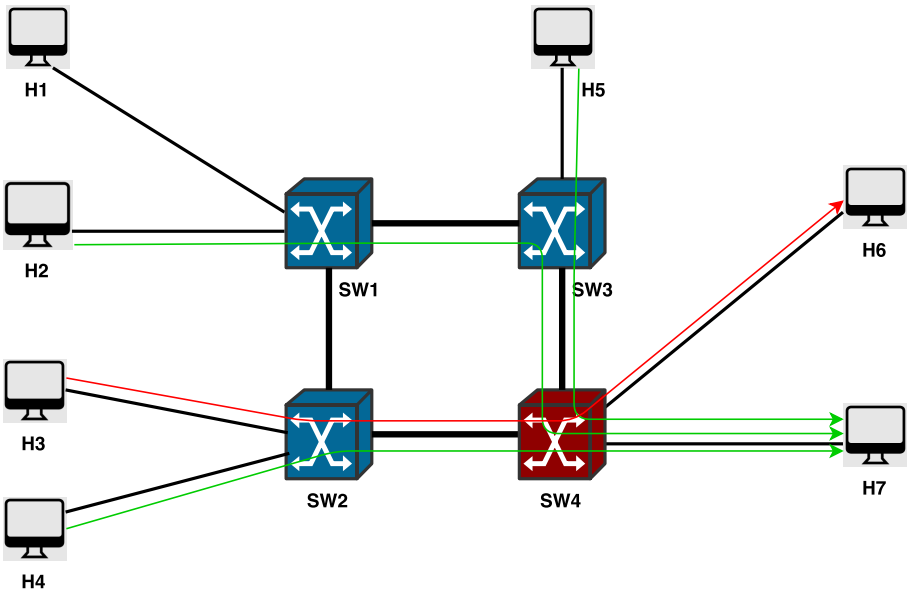

Figure 2.8: Victim Flow.

Here, we can say H3 is experiencing *Head-of-Line(HoL) blocking*. HoL blocking is a network performance issue that occurs when a bunch of packets are blocked by the first packet in line. This situation usually happens in input buffered network switches whenever traffic waiting to be transmitted prevents or blocks traffic destined elsewhere from being transmitted. The source node H3 is a *victim* of congestion and the associated flow is know as *victim flow*. Moreover, as the victim node H3 is blocked at SW2, traffic pills up towards H3 as well. So, the congestion tree grows towards the victim node H3, which indicated with red line, as shown in Figure 2.9. In this way, the effect of HoL blocking may spread towards the whole network in larger topologies although its effect is not visible for this small topology.

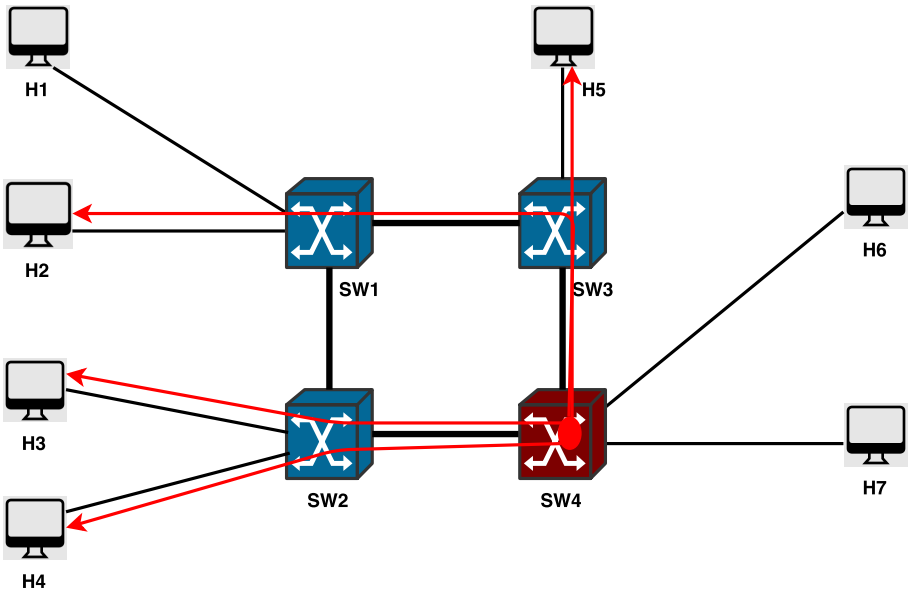

Figure 2.9: Congestion Tree for Victim Flow.

There are several mechanisms that are used to avoid congestion. In *lossy* networks such as the Internet, discarding packets in congestion situations is allowed to avoid congestion. However, packet dropping and retransmission are not permitted in *lossless* InfiniBand high performance networks. The simplest way to overcome congestion in lossless networks is overprovisioning the network, which means providing excess amount of network resources to the network so that congestion will not occur. However, the network should be designed to handle the maximum possible amount of traffic to effectively utilize the network resources. As a result, this mechanism is not a cost effective and scalable solution in recent HPC clusters and data centers.

Adaptive routing and load balancing are the other types of techniques to solve the problem of congestion [10, 12]. With these strategies, the status of the network is examined to make routing decisions and then adapted to it by choosing a path that is underutilized, as we discussed in Section 2.1.2. However, to implement such techniques, the network should have enough alternative paths that avoid the root of the congestion tree. As a result, these techniques will not be a good solution for congestion rooted from the downstream nodes at which there is no alternative paths. In addition, these techniques cannot avoid network performance degradation once congestion occurs. Furthermore, since users are free to implement any type of network topology, the requirements of these strategies may not be met by the InfiniBand network.

In general, as stated in [9, 22, 44], an efficient congestion control mechanism should have the following characteristics in order to provide more efficient use of network resources and more flexible bandwidth allocation schemes:

- *Efficient utilization of network bandwidth*: the congestion control

mechanism should not be too aggressive to reduce the injection of traffic beyond what is necessary to avoid underutilization of resources.

- *Fairness*: traffic flows sharing a given resource should have the same throughput irrespective of congestion.
- *Fast reaction*: the congestion control mechanism should respond as fast as possible to changes in the traffic flow. It should increase and decrease the injection rate of traffic as quickly as possible to avoid underutilization of resources and congestion spreading, respectively.
- *Low oscillation*: the congestion control mechanism should provide stability in face of high bandwidth or large delay. In addition, it should maintain a proper balance between fast reaction and low oscillation on the injection rate of flows while maximizing the network bandwidth utilization.

2.3 InfiniBand Architecture

2.3.1 InfiniBand Concepts

InfiniBand is a switch-based point-to-point network communications standard that defines the communication protocol between processing nodes and Input/Output devices. The InfiniBand Architecture (IBA) [35, 20] is developed for today's high performance systems with the ability to scale for next generation system requirements. The InfiniBand's ability to carry multiple traffic types over a single connection makes it ideal for clustering, communications and storage systems. As a result, the interconnect technology is used in many data centers, HPC clusters, storage and embedded applications that scale from two nodes to a single cluster of tens-of-thousands of nodes to provide reliability, availability, performance, and scalability necessary for present and future server systems. InfiniBand is designed to produce leading performance with the highest efficiency to fully utilize data center infrastructures and solve the lack of high bandwidth, concurrency and reliability of existing technologies for system area networks whose topology can be established by the users.

In recent times, high performance clusters are based on different available network technologies such as Gigabit Ethernet, 10 Gigabit Ethernet, Fibre Channel, Myrinet, ServerNet, Autonet, etc [53, 11]. However, the InfiniBand standard has received an increasing attention since it provides the protection, isolation, deterministic behavior, and quality of services required by high-speed interconnect technology. The dominance of InfiniBand in the world's most powerful supercomputers is due to the standard's superior performance, efficiency and scalability. TOP500 list result [42] revealed that InfiniBand connects 45% of the 73 most powerful Petascale systems. Additionally, almost half of the systems on the list are used for Research or in Academic institutions, industries that require high performance and high efficiency. InfiniBand's high

bandwidth, low latency and CPU offloads make it the preferred solution for these systems.

InfiniBand is a switched fabric communications link primarily used in high-performance computing. It is a switch-based standard interconnect which allows switches to support up to 15 *Virtual Lanes* per port (Port VL) for data traffic and exclusively reserved one additional VL for subnet management. InfiniBand provides a direct access to an easy-to-use messaging service for every applications, i.e. an application need not rely on the operating system to transfer messages. The messaging service can be used to communicate with other applications or processes or to access storage. InfiniBand provides the messaging service by creating a channel connecting an application to any other application or service with which the application needs to communicate. This mechanism of the InfiniBand is totally different from the standard TCP/IP network stack where accessing the shared network resources are solely owned by the operating system. However, InfiniBand comes with its own challenges with regard to congestion and congestion control mechanisms. Congestion and congestion spreading in InfiniBand are caused by the use of link level credit-based flow control, as we discussed in Section 2.1.3. As a result, congestion, congestion control, the IB CC mechanisms, and the parameter values of the IB CC mechanisms need a wide range of research.

2.3.2 Congestion Control in InfiniBand

Congestion Control is a counter measure for Head-of-Line(HoL) blocking experienced in interconnected networks due to congestion spreading as shown in Section 2.2. It is used to avoid congestive collapse by reducing the injection rate of packets in to the network. The InfiniBand congestion control (IB CC) mechanism is based on a closed-loop feedback control system, where a forwarding node detecting congestion is responsible for informing all nodes that are injecting traffic and contributing to congestion about the situation [13, 17, 36]. Yoshio Turner et. al [44] proposed a three-stage IB CC mechanism which relies on end-to-end Explicit Congestion Notification (ECN) packet marking to resolve congestion with traffic injection rate throttling. The purpose of ECN-based injection throttling strategy is reducing HoL blocking by removing the source of the congestion. In the forward explicit notification approach, packets are marked as contributing to congestion by setting a specific bit in the packet header. The ECN is the best method for congestion control in InfiniBand since it is completely dynamic and requires little knowledge about the network or traffic patterns that are expected. Figure 2.10 illustrates how a three-stage IB CC mechanism operates.

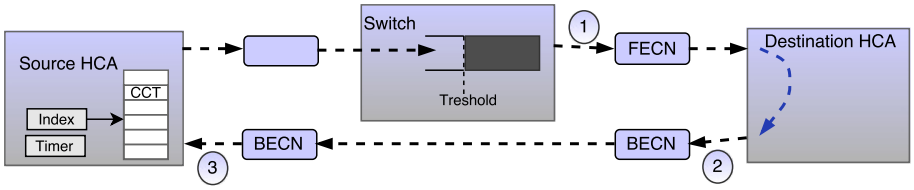

Figure 2.10: InfiniBand Congestion Control Operation.

As shown in Figure 2.10, the IB CC mechanism works as follows: a switch detects congestion when its buffer starts to fill up and reaches at a certain defined threshold. When the switch detects the congestion, the packet coming from the source node which contributes to the congestion will be marked by setting a specific bit, known as *Forward Explicit Congestion Notification* (FECN), in the packet header (Figure 2.10 ①). Once the packet is marked with FECN bit, it will be sent to the destination node called *HCA*. The HCA that receives the packet with FECN bit, registers the FECN bit and responds to the source HCA with a packet marked with another bit called *Backward Explicit Congestion Notification* (BECN) (Figure 2.10 ②). The source HCA receiving a packet marked with BECN bit is expected to lower its injection rate of the corresponding traffic flows to avoid the congestion (Figure 2.10 ③). However, this injection throttling should only be sufficient enough to remove the congestion, and should not be too aggressive as it may leave the resources at the root of the congestion idle. As a result, after congestion is removed, the source node will again gradually increases the traffic injection rate to avoid underutilization of the network resources, which may again cause congestion to be occurred.

There are several parameters [15], in both the switch and the channel adapter, that determine the behaviour of the IB CC mechanism. The values of these set of parameters are controlled by the *Congestion Control Manager* (CCM). The network characteristics such as when the switch detects congestion, at what rate the switch will notify the destination nodes using the FECN bit, and how much and for how long a source node contributing to congestion will reduce its injection rate are all determined by the values of these parameters. If all the values of the parameters are appropriately set, the network can resolve congestion, avoid head-of-line blocking, and utilize its resources efficiently. Table 2.1 shows the list of congestion control parameters on both the switch and the channel adapter.

Switch	Channel Adapter
Threshold	Congestion Control Table (CCT)
Marking_Rate	CCTI (CCT_index)
Packet_Size	CCTI_Increase
Victim_Mask	CCTI_Timer
	CCTI_Limit
	CCTI_Min

Table 2.1: InfiniBand Congestion Control Parameters

Switch IB CC Parameters:

The switch is responsible for detecting congestion and notifying the destination nodes by forwarding packets marked with FECN bit. Congestion is detected at a switch when a given threshold value is crossed at a given Port VL. The *Threshold* values are ranging from 0 to 15, where 0 indicates that no packet marking and 15 is the low threshold with least probability of congestion. If the Port VL is the root of congestion while having available credits to output data, it enters into a congestion state. On the other hand, if Port VL is in a congested state and it has no available credits, it is considered to be a victim of congestion. Although a Port VL has no available credits, it may enter into a congested state and considered to be the root of congestion if a specific *Victim_Mask* is set to the port. A *Victim_Mask* is usually set for a switch port that is directly connected to a host node. A host node which is not fast enough to process the received packets will still not consider itself to be the root of congestion. Packet marking is performed based on two CC parameters: *Packet_Size* and *Marking_Rate*. A packet with a smaller size than the specified *Packet_Size* won't be marked with FECN bit. *Marking_Rate* specifies the average number of packets sent unmarked between two consecutive marked packets. Generally, as summarized in Table 2.1, *Threshold*, *Marking_Rate*, *Packet_Size* and *Victim_Mask* are the basic IB CC parameters at a switch.

Channel Adapter IB CC Parameters:

A destination HCA that receives FECN bit forwarded from the switch detecting the congestion will return a BECN bit to the source HCA in order to lower the injection rate of the corresponding traffic flow as soon as possible. Up on receiving the BECN bit, the source HCA attempts to determine by how much and for how long the injection rate should be reduced by using *Congestion Control Table* (CCT). The CCT contains an array of values of injection rate delay (IRD) which used to control congestion. The data is organized such that the lowest IRD is contained in entry 0, and the highest IRD is contained in the last entry of the table. Each congested traffic flow will have an index, *CCTI*, in the CCT. Whenever a new BECN arrives, the CCT Index of the flow is increased by *CCTI_Increase* util it reaches to an upper bound limit *CCTI_Limit*.

The lowered injection rate of the traffic flow should increase again after a certain specified time to avoid underutilization of resources at the root of the congestion. The HCA uses the *CCTI_Timer* with each flow that is going to be expired together with a decrement in one CCTI. When the CCTI of a flow reaches zero, the flow no longer experience any injection rate delay. However, it is possible to enforce a minimum injection rate delay using *CCTI_Min* since CCTI won't be less than the *CCTI_Min*. Overall, as as listed out in Table 2.1 above, *CCT*, *CCTI_Increase*, *CCTI_Limit*, *CCTI_Timer* and *CCTI_Min* are the basic parameters at the HCA.

2.4 OMNeT++

OMNeT++ stands for Objective Modular Network Testbed in C++ [4]. OMNeT++ [32] is an extensible, modular, component-based C++ simulation library and framework, primarily for building network simulators. It is an open source discrete event simulation environment that can be used under the Academic Public License [47] that makes the software free for non-profit use. OMNeT++ is not a network simulator but a framework to allow us to create our own network simulations. OMNeT++ aims at providing a rich simulation platform, and leaves creating simulation models to independent research groups. The simulation platform is designed for modeling communication networks, multiprocessors and other distributed or parallel systems [47]. OMNeT++ has large user community in the areas like academic, educational and research institutions.

OMNeT++ offers an Eclipse-based IDE, a graphical runtime environment, and a host of other tools to create different components of the simulation such as networks, modules (simple and compound) and channels. To define the structure of the simulation network topology model in OMNeT++, we need to define the modules and their interconnections using the OMNeT++ topology description language called *NED* [47]. NED is a network description language that is used to create the topology of networks in OMNeT++. In the NED file, the interfaces of the module such as *gates* and *parameters*, interconnection among switches and between switches and HCAs, positions of the nodes and network definitions will be defined. The OMNeT++ IDE allows to define the network topology either graphically or using NED as its native file format and allows to switch between the two views at any time. The network topology can be created and edited using the editor either graphically or in NED source view using hand-written NED code. If we create our network topologies using the graphical editor, the coinciding NED source code will also be generated for us. The choice to use the graphical editor over the NED language is purely ours.

Mellanox Technologies which is the leading provider of InfiniBand technologies has released an open-source InfiniBand simulation model for OMNeT++ community [31]. Based on this IB model, Ernst Gunnar Gran and Sven-Arne Reinemo. [13] implemented a Congestion Control capable IB model in the OMNeT++ environment. The CC capable IB model consists of the implementation of HCAs and switches with routing tables and

virtual lanes. The model has support for all switch's and HCA's IB CC parameters and implemented according to the IB CC specifications [20]. In this thesis, we will use the CC capable IB model in the OMNeT++ simulation environment.

Chapter 3

Approach

This Chapter presents a brief explanation about the necessary actions that will be taken to address the problem statement described in Section 1.1. It conveys a concrete feeling on how the thesis is conducted and provides a clear illustration on the terminologies, techniques, experiment designs, technologies and tools that will be used in this work. The chapter is organized in six sections and provides detailed information for an experienced investigator to the extent that one can replicate the study.

3.1 Objective

Network and system administrators are highly interested in performance evaluations since their objective is to provide the highest performance at the lowest cost. Consequently, performance evaluation has become a continues process that helps to determine the relevant settings of a given system that maximizes its efficiency. However, exploring the parameter spaces and evaluating the performances of large scale clusters very expensive since the system consist of thousands of processing nodes interconnected by a network. As a result, a simulation platform which can generate a realistic synthetic workloads whose statistical properties match those of traffic observed on a real network link is needed. The InfiniBand model has packet generator to generate traffic for InfiniBand network simulations in OMNeT++ platform.

In this work, a simulation-driven exploration of the parameter spaces of IB CC mechanisms which verify the performance gains of the mechanism and ensure it can in fact work properly will be conducted. The main goal of this thesis is to explore the appropriate set of IB CC parameters that are discussed in Section 2.3.2 by systematically running experiments on different scenarios using 2D and 3D mesh network topologies.

In doing so, the following major activities will be made:

- Developing scripts that can create network topologies for the InfiniBand model in OMNeT++ environment.
- Developing scripts that can populate the InfiniBand switches with a forwarding table entry based on the dimension order routing

algorithm.

- Designing a series of experiments and setting up the simulation environment.
- Configuring and tuning IB CC parameters through simulations with OMNeT++.
- Collecting data and plotting graphs for better analysis.
- Analysing and evaluating the performance gains of congestion control for a given set of parameters and determine the one that results the best performance.

3.2 Simulation Environment Setup

The experiment will be designed and conducted on the ALTO Openstack cloud [33]. ALTO is a computing cloud owned by *Oslo and Akershus University College of Applied Sciences*, which allows teachers and students to create and user their own virtual computers. The first step that will be made on the ALTO cloud is uploading Ubuntu Desktop 14.04 LTS image [7] to get the support of OMNeT++ IDE environment. The OMNeT++ IDE environment is required because we need to get the benefit of the *Tkenv* graphical run-time interface which will enable us to easily understand how the simulator works. Tkenv will help us to verify, animate, debug, easily understand and execute the simulation at the early stage of the simulation setup [46].

Initially, an instance which serve Ubuntu Desktop 14.04 operating system with a large flavor, 8GB RAM, 4 VCPU and 80GB Disk will be created using the uploaded image. Then OMNet++ 4.6 simulator [32] and other prerequisite packages required for OMNeT++ will be installed on this VM. The package will be built and all appropriate configurations such as setting environment variables and workspaces will be done for the simulator. After confirming that OMNeT++ is running and working properly, the InfiniBand model (IB Model) which support congestion control will be imported to the simulator and built successfully. Finally, a project which consists of the required *INI* and *NED* files as well as forwarding tables for our simulation topology model will be created. INI files contain the configuration of the simulation model with *.ini* extension for simulation runs in OMNeT++. NED files are *.ned* extension files used to describe the structure of a simulation model. These files are explained in detail in Section 2.4.

Since the study requires a large number of simulation runs, a single VM won't be enough to conduct all the simulations required within this short project period. To alleviate this situation, we need to create other instances which have similar configurations and tools with the first one. The simplest solution for this is to take the snapshot of the first VM and create a number of VMs less or equal to the quota allowed for our project on the ALTO cloud using this snapshot image. For our case, we will create *four* other instances

with the same flavor. As a result, a total of *five* instances will be used for the simulation.

To further increase the speed of the simulation, instead of using the OMNeT++ IDE environment (*Tkenv*), *Cmdenv* will be used for simulation. So, the simulation will be easily run from a command line using the following command:

Running the Simulation from the Command Line

```
../../ibmodel -r 0 -u Cmdenv -f TwoDMeshTopo.ini -c flood
-n /home/ubuntu/ibmodel/
```

where, *ibmodel* is the simulation model (ibmodel) executable file, *TwoDMeshTopo.ini* is the main configuration file, *flood* is the configuration section, and */home/ubuntu/ibmodel/* is the the location of the *ned* files.

3.3 Network Topology Design

In M-dimensional mesh network, every switch is connected to n number of HCAs and $2M$ number of its neighboring switches (except boundary switches) [24]. Thus, the degree of a non-boundary switch in an M-dimensional mesh is $2M$. The number of physical connections per switch remains fixed in a mesh network even if the size of the network increases. Due to the large number of nodes (HCAs and Switches) and links in such type of networks, designing the network architecture in OMNeT++ environment is challenging.

In our simulation network topology model, it will be very challenging to create the topology using either the graphical interface or the source code for every scenario due to the number of switches and nodes used for our simulation will be large and a relatively more interconnection is required in mesh and torus topologies. Therefore, it is more efficient, practical and flexible to develop scripts which can create the simulation network topology model with M number of switches in one dimension of a square mesh and torus topology with N number of HCAs connected at each switch. With an aim to work on 2D and 3D mesh as well as 2D torus topology, three python scripts that are capable of generating these topologies will be developed. The mechanisms how those scripts work and the assumptions made will be explained in this section.

3.3.1 2D Mesh Topology Design

In 2D mesh network architecture, every switch is connected to n number of HCAs and *four* of its neighbor (except the boundary switches) switches as shown in Figure 3.1. The main challenge while creating the interconnection is therefore determining the input/output ports of the switches such as how many ports should a given switch has? and which port of a switch is connected to which neighbouring node? In the OMNeT++ IB Model, if a port is created for a switch, it has to be used (connected to a node) otherwise the error will be generated. The number of ports of a switch will

be determined by the number of HCAs connected to the switch plus the position of the switch in the topology. The problem associated with the number of ports connected to the HCAs can be easily solved by attaching equal number of HCAs at each switch. As shown in Figure 3.1, *n* number of HCAs are attached to each switch at ports from *0* to (*n-1*). In the diagram, we have only shown the HCAs at the corner switches and the switches at the corner, the middle of the edges and the non-boundary switches located at the center of the topology to avoid complexity on the drawing. However, it should be visualized as all the switches in the topology have equal *n* number of HCAs attached to them and there are switches at the dashed lines of the figure. The dashed lines between HCAs doesn't indicate that a link exists between HCAs rather they show that there may be *n* number of HCAs attached to a switch.

Number of ports

To determine the exact number of ports of a given switch, in addition the number of HCAs attached to it, the position of the switch in the topology must be known. As shown in Figure 3.1, in any 2D mesh topology, a switch may take either of the *three* positions, it may be at the corner, at the edge, or at the middle of the topology. Every switch at the corner should be connected to *two* other switches, one in the *x* and the other in the *y* direction. Switches at the edges of the topology should be connected to *three* other switches. The switches in the middle should be connected to *four* other switches, two in the *x* and two in the *y* direction. Therefore, for *n* number of HCAs attached to the switches, corner switches will have (*n+2*) ports, switches at the edges will have (*n+3*) ports and switches at the middle will have (*n+4*) ports. As shown in Figure 3.1, switches at the corner connected to other switches at ports *n* and (*n+1*), switches at the edges are connected at ports *n* through (*n+2*) and switches at the middle are connected at ports *n* through (*n+3*) to other neighbouring switches.

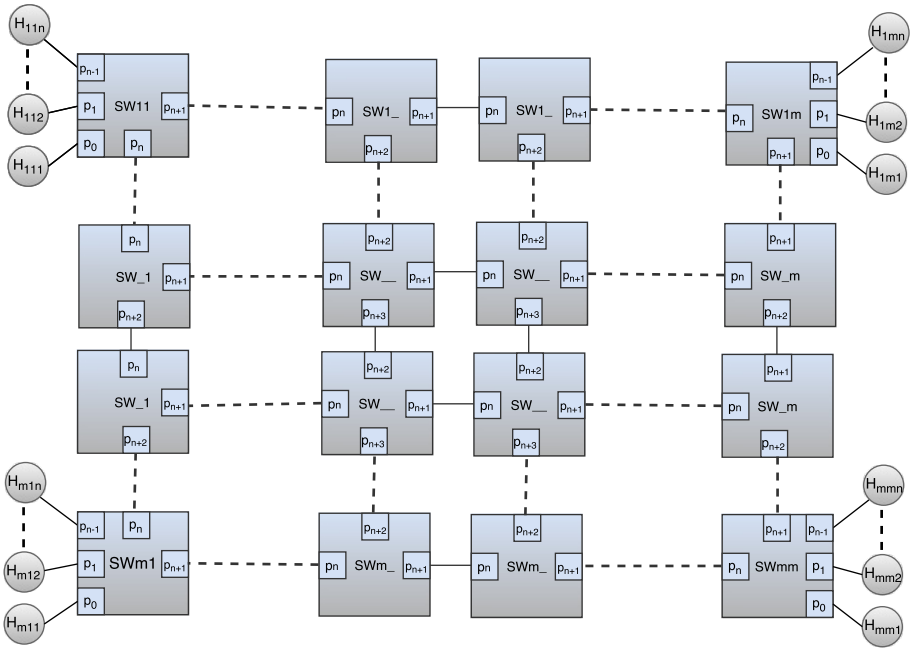

Figure 3.1: Generalized 2D Mesh Topology.

Naming

In our case, as a convention, the switches are named starting from the top-left corner of the network topology, and then progressing towards the right by increasing the second subscript. Consequently, for *mxm* 2D mesh topology, a switch at the top-left will be named *SW11*, a switch at the top-right corner will be named *SW1m*, a switch at the bottom-left corner will be named *SWm1*, a switch at the bottom-right corner will be named *SWmm* and so on. HCAs attached at each switch will be named by replacing *SW* to *H* and adding a third subscript to the switch subscript attached to it. As shown in Figure 3.1, for n number of HCAs attached at each switch, a subscript from 1 to n will be used to name the HCAs together with the switch subscripts. For instance, for n number of HCAs attached at top-left switch, *SW11*, the name of the HCAs will be *H111, H112*, and so on until *H11n*.

Connection

All the switches at the topmost and bottommost edges, except the corner switches, will be connected to a switch at its left with port number *n* and to a switch to its right side with a port number (*n+1*), and to a switch at its bottom or upper switch with port number (*n+2*). Switches at the leftmost edges are connected with a port number *n* to a switch at its top, a port number (*n+1*) to the right and a port number (*n+2*) to a switch at its bottom. Switches at the rightmost edges are connected with a port number *n* to a switch at its left, a port number (*n+1*) to its top and a port number (*n+2*) to a switch at its bottom. All switches at the middle will be connected with

n port to the left, (*n+1*) port to the right, (*n+2*) port to up and (*n+3*) port to down.

3.3.2 2D Torus Topology Design

The existence of wrap around links in torus adds extra complexity on the design of the topology on OMNet++ IDE environment. On the contrary, writing a script for 2D torus topology is relatively easier since all switches have equal number of ports, (*n+4*), where *n* is the number of HCAs attached to a switch. We used a similar argument as in the above to name the switches and HCAs. The only difference here is we need to add 1 port for each switch at the edges of the topology and 2 ports for each corner switches, and adding wrap around links for the ports. Which means that every switch in 2D torus will have a similar structure with switches at the middle of the 2D mesh topology. Therefore, all switches in 2D torus will be connected with its *n* port to the left, (*n+1*) port to the right, (*n+2*) port to up and (*n+3*) port to down. The first *n* ports(from 0 to *n-1*) of each switch will be attached to n HCAs, as shown in Figure 3.2

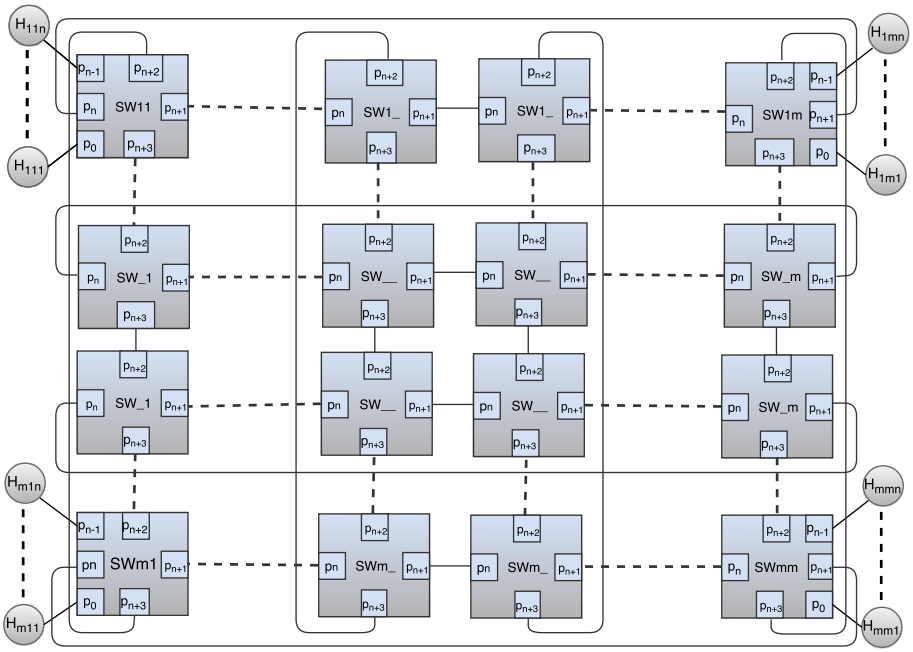

Figure 3.2: Generalized 2D Torus Topology.

3.3.3 3D Mesh Topology Design

In 3D mesh network architecture, every switch is connected to *n* number of HCAs and *six* of its neighbor (except the boundary switches) switches. Similar to 2D mesh topology, the number of ports of a switch in 3D topology will be determined by the number of HCAs attached to the switch and its position in the topology. However, finding the position of a switch in 3D is a complicated process. As shown in Figure 3.3, a switch may be

located at the boundary or internal to the network topology (non-boundary switches). The boundary switches are located on the surface of the "3D cube" created by the topology. In any of the surfaces, the boundary switches may be located at their corners, edges or internal to the surface.

Number of ports

The exact number of ports of a switch in 3D mesh is also determined by the number of HCAs connected to the switch and the position of the switch in the topology as in 2D mesh topology. In 3D mesh, there are *eight* corner switches which are connected to *three* of its neighbor switches. A switch at the edges of each plane is connected to *four* of its neighbor switches, and switches at the middle of each plane are connected to *five* of its neighbor switches. Non-boundary switches are connected to *six* of its neighbor switches as mentioned above. As a result, corner switches have (*n+3*) ports, switches at the edges of a plane have (*n+4*) ports, switches at the middle of each plane have (*n+5*) ports and non-boundary switches have (*n+6*) ports, where *n* is the number of HCAs attached to a switch.

Naming

We started naming the switches from the top-left corner of the topology, such that the top-left corner switch is *SW111*, and progressing towards the positive z-direction by incrementing the third subscript and then to the positive x-direction by incrementing the second subscript. As shown in Figure 3.3, for a cubic *mxmxm* 3D mesh topology, the 8 corner switches will be named *SW111*, *SW11m*, *SW1m1*, *SW1mm*, *SWm11*, *SWm1m*, *SWmm1* and *SWmmm*. The names of HCAs will be derived from the switches attached to them. For instance, HCAs attached at *SWmmm* will be named *Hmmm1*, *Hmmm2*, and so on till *Hmmmn*.

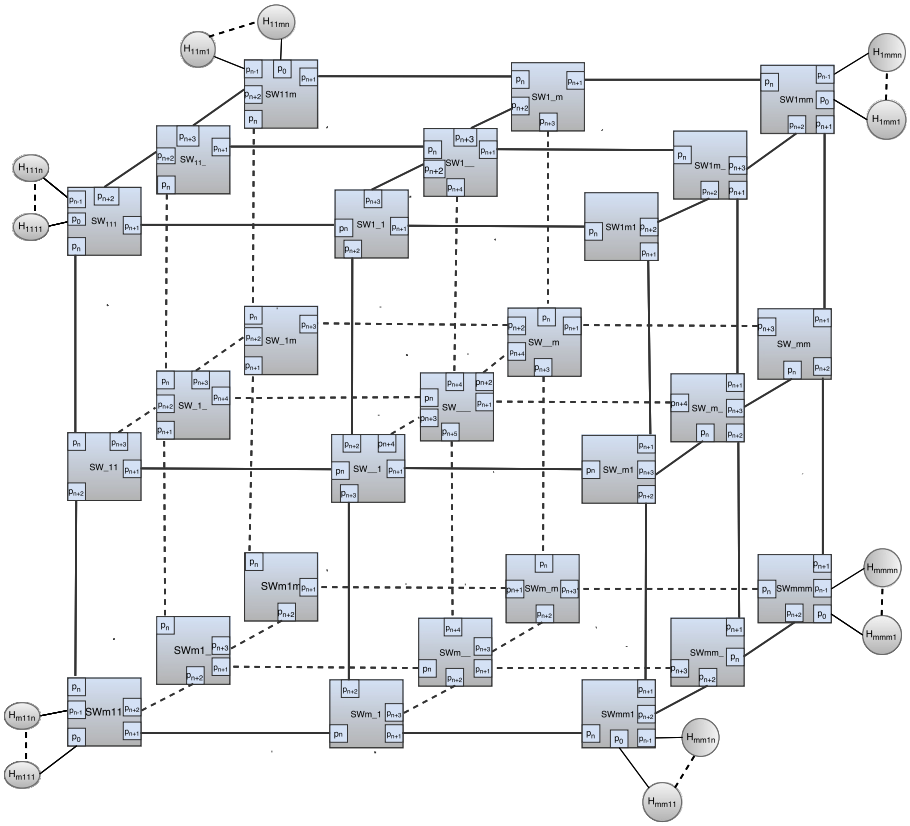

Figure 3.3: Generalized 3D Mesh Topology.

Connection

In 3D mesh topology, establishing a link between switches with a specified port is challenging and needs thorough look at in to it. In this network architecture, there are 6 faces (top and bottom xz-planes, front and back xy-planes and left and right yz-planes), which results in 12 edges, 8 corners and switches internal to the planes. They are collectively called *boundary* switches. The rest of the switches are called *non-boundary* switches which are located internal to the cube. Each switch type needs different treatment based on their orientation during connection establishment.

In larger 3D mesh topologies, the majority of the switches will be non-boundary switches which have the same number of ports (*n+6*) and can be easily generalized. In our case, all non-boundary switches, such as *SW___* in Figure 3.3, (*SW333 where m=3*), is connected with a port *n* to the negative x, (*n+1*) to the positive x, (*n+2*) to the positive z, (*n+3*) to the negative z, (*n+4*) to the positive y and (*n+5*) to the negative y directions. As in 2D mesh above, HCAs will be attached to a switch from port 0 to (*n-1*). The boundary switches at different locations need a particular treatment. However, once the interconnection boundary switches is carefully defined, it can be used for any large topology. Tables 3.1, 3.2 and 3.3 summarizes the interconnection of switches located at the corners, edges and internally to the surface of the cube, respectively.

Switch Position	+ve X	-ve X	+ve Y	-ve Y	+ve Z	-ve Z
SW111	n+1	-	-	n	n+2	-
SW11m	n+1	-	-	n	-	n+2
SW1mm	-	n	-	n+1	-	n+2
SW1m1	-	n	-	n+1	n+2	-
SWm11	n+1	-	n	-	n+2	-
SWm1m	n+1	-	n	-	-	n+2
SWmm1	-	n	n+1	-	n+2	-
SWmmm	-	n	n+1	-	-	n+2

Table 3.1: 3D Mesh port-wise Interconnection for Switches at the Corners

Switch Position	+ve X	-ve X	+ve Y	-ve Y	+ve Z	-ve Z
Edges b/n SW111 & SW11m	n+1	-	-	n	n+3	n+2
Edges b/n SW111 & SW1m1	n+1	n	-	n+2	n+3	-
Edges b/n SW11m & SW1mm	n+1	n	-	n+3	-	n+2
Edges b/n SW1m1 & SW1mm	-	n	-	n+1	n+3	n+2
Edges b/n SW111 & SWm11	n+1	-	n	n+2	n+3	-
Edges b/n SW11m & SWm1m	n+3	-	n	n+1	-	n+2
Edges b/n SW1mm & SWmmm	-	n+3	n+1	n+2	-	n
Edges b/n SW1m1 & SWmm1	-	n	n+1	n+2	n+3	-
Edges b/n SWm11 & SWmm1	n+1	n	n+2	-	n+3	-
Edges b/n SWm11 & SWm1m	n+1	-	n	-	n+3	n+2
Edges b/n SWm1m & SWmmm	n+3	n+1	n	-	-	n+2
Edges b/n SWmm1 & SWmmm	-	n+3	n+1	-	n	n+2

Table 3.2: 3D Mesh port-wise Interconnection for Switches at the Edges

Switch Position	+ve X	-ve X	+ve Y	-ve Y	+ve Z	-ve Z
Top xz-plane	n+1	n	-	n+4	n+3	n+2
Bottom xz-plane	n+1	n	n+4	-	n+3	n+2
Front xy-plane	n+1	n	n+2	n+3	n+4	-
Back xy-plane	n+1	n+2	n	n+3	-	n+4
Left yz-plane	n+4	-	n	n+1	n+3	n+2
Right yz-plane	-	n+4	n+1	n+2	n+3	n

Table 3.3: 3D Mesh port-wise Interconnection for Switches internally to the Planes

3.4 Routing and Forwarding Tables

Similar to the interconnection topology, the routing scheme is also the fundamental element that has to be considered while evaluating the performance of an interconnection network. As we have discussed in Section 2.1.2, InfiniBand networks use deterministic routing. For 2D mesh networks, a deterministic dimension order routing algorithm, called *DOR*, which routes packets first in x-direction to the correct column and then in y-direction to the receiver is often used [41]. In 3D mesh, DOR works similar to 2D such that it routes packets first in x-direction, next in y-direction, then in z-direction to the receiver. This routing algorithm fits well for mesh and torus topology networks.

In InfiniBand network, each processing node has an associated local identifier (LID). In addition, since InfiniBand networks use packet switching technique, each switch has to maintain their on forwarding table. The purpose of the forwarding table is mapping the LID in the destination LID (DLID) field of a packet to an output port of the switch to which the packet has to be forwarded [27]. As a result, the forwarding table contains the output port to be used at the switch for each possible destination. The sender puts the address of the receiver, the DLID, to the header of the packet at the start of the routing. Then, when a packet with a given DLID arrives at a switch, the packet will be forwarded to the corresponding output port via the forwarding table lookup. That means packets are routed at each switch by looking into the forwarding table entry.

To run the simulation, the forwarding table entry holding port groups for each switch together with the switch indexes should be feed to the simulator. In large 2D and 3D mesh topologies, setting this table entry manually is a tedious and time consuming task as well as it is prone to error. To alleviate this problem, a script which can generate a Forwarding Databases (FDBs) file and populates each switch with the output port

numbers for each HCAs based on the DOR routing algorithm will be developed and used for 2D and 3D mesh networks. The output port used for each LID in a given switch will be based on the port-wise interconnection of the switches in the topology used in Section 3.3.

3.5 Automation Tools

This section presents *five python scripts* developed to automate the creation of topologies and forwarding tables, and their results in the OMNeT++ environment. The first three scripts (*2DMeshTopo.py, 2DTorusTopo.py* and *3DMeshTopo.py*) were developed to generate topology source codes for 2D mesh, 2D torus and 3D mesh topologies, respectively. The scripts were able to generate **.ned* file source codes that can be visualized and used in the OMNeT++ IDE environment. The last two scripts were developed to produce forwarding tables. In order to populate forwarding tables of the switches based on a deterministic DOR routing algorithm, two scripts, *2DMeshRouting.py* and *3DMeshRouting.py* that can generate forwarding databases (FDBs) for 2D and 3D mesh topologies have been developed. The scripts were developed based on the interconnection principles used for the topology.

3.5.1 Script: *2DMeshTopo.py*

Based on the assumptions and explanations made in Section 3.3 for the naming, number of ports and port-wise interconnections of 2D mesh topology, a python script known as *2DMeshTopo.py* has been developed to generate source codes in the **.ned* file format. The python script accepts the values m and n, where *m* is the number of switches that will be in one dimension of a square 2D mesh topology and *n* is the number of HCAs at each switch, and then creates source codes for an *mxm* 2D mesh topology. Figure 3.4 shows an example of a 2D mesh topology created on the OMNeT++ IDE environment using the script when *m=3* and *n=2*. For this simple 3x3 2D mesh topology, there are 9 switches, 18 HCAs, 1 generator, and 48 bidirectional links (including the links to the generator). This indicates that creating larger topology either graphically or source editing using the OMNeT++ IDE will be very complex unless we used such automation.

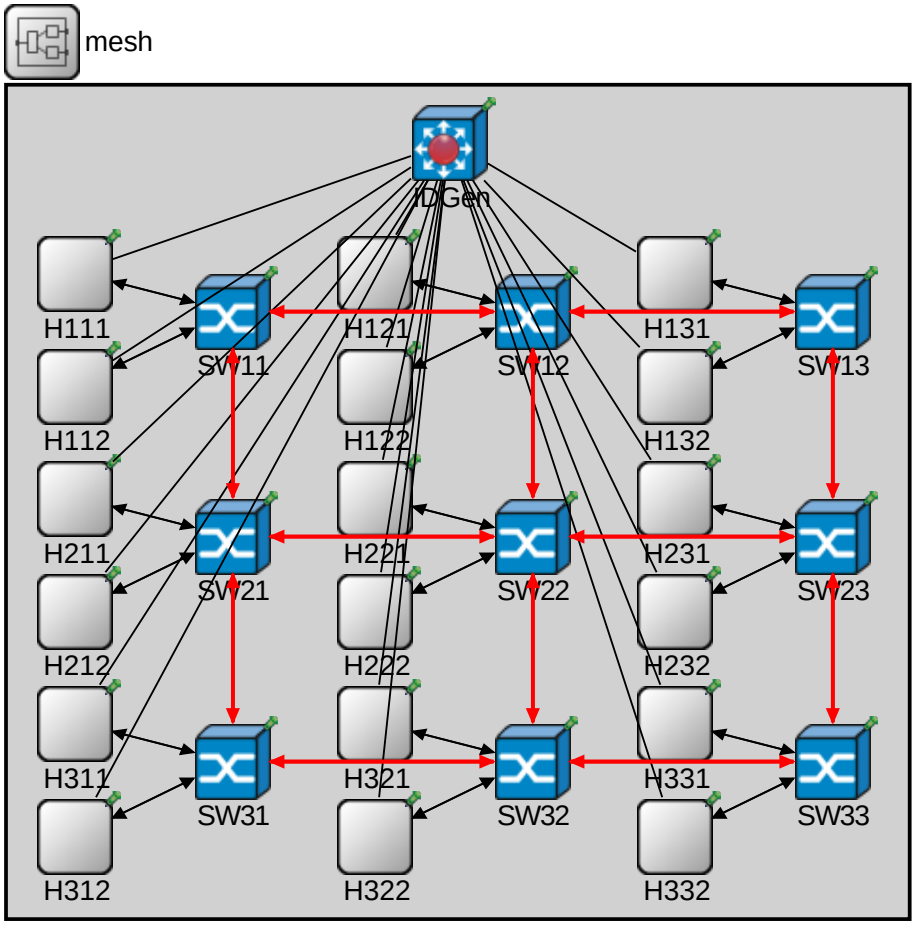

Figure 3.4: 2D Mesh Topology in OMNeT++ Environment.

3.5.2 Script: *2DTorusTopo.py*

This script has been written for the purpose of generating *ned* file source code for 2D torus topology. Similar to the script developed for 2D mesh, this script also accepts any two numeric values m and n from the user for number of switches in one dimension of 2D torus and number of nodes attached to each switch, respectively. Figure 3.5 shows an example of a 3x3 2D torus topology created on the OMNeT++ IDE environment using this script. Even though, they are not clearly visible on the diagram because of overlapped links, there are 6 more links in this torus topology than the one shown in Figure 3.4 above. The red lines which cross over the middle switches show that there are wrap around links between the end switches both vertically and horizontally.

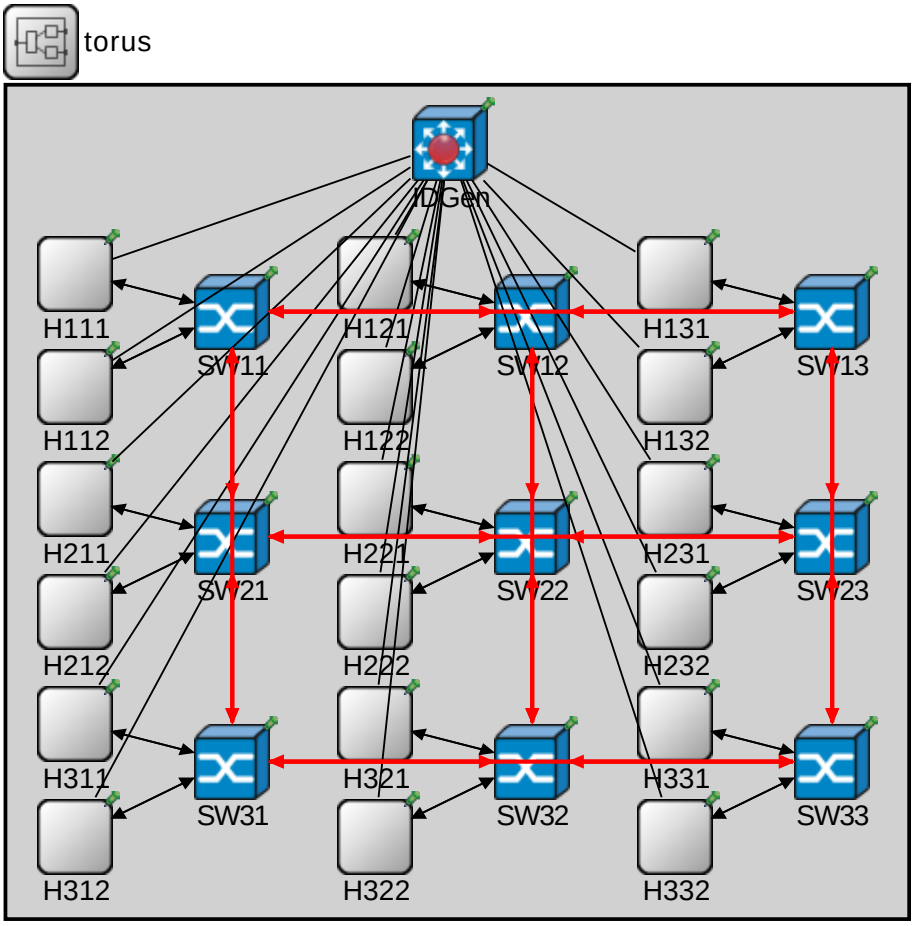

Figure 3.5: 2D Torus Topology in OMNeT++ Environment.

3.5.3 Script: *3DMeshTopo.py*

Using the knowledge explained in the approach chapter, a script that accept any numeric values m and n, where m is the number of switches in one dimension of 3D mesh topology and n is the number of switches attached to each switch, has been developed. Figure 3.6 demonstrates the result of the script for m=3 and n=2 in the OMNeT++ environment. In this simple 3x3x3 mesh topology, there are 27 switches, 54 HCAs, 1 generator and several number of port specific links. This indicates that for larger 3D mesh topologies, it is almost impossible to design a topology in the OMNeT++ IDE with either a simple drag and drop in the design view or *ned* source code writing.

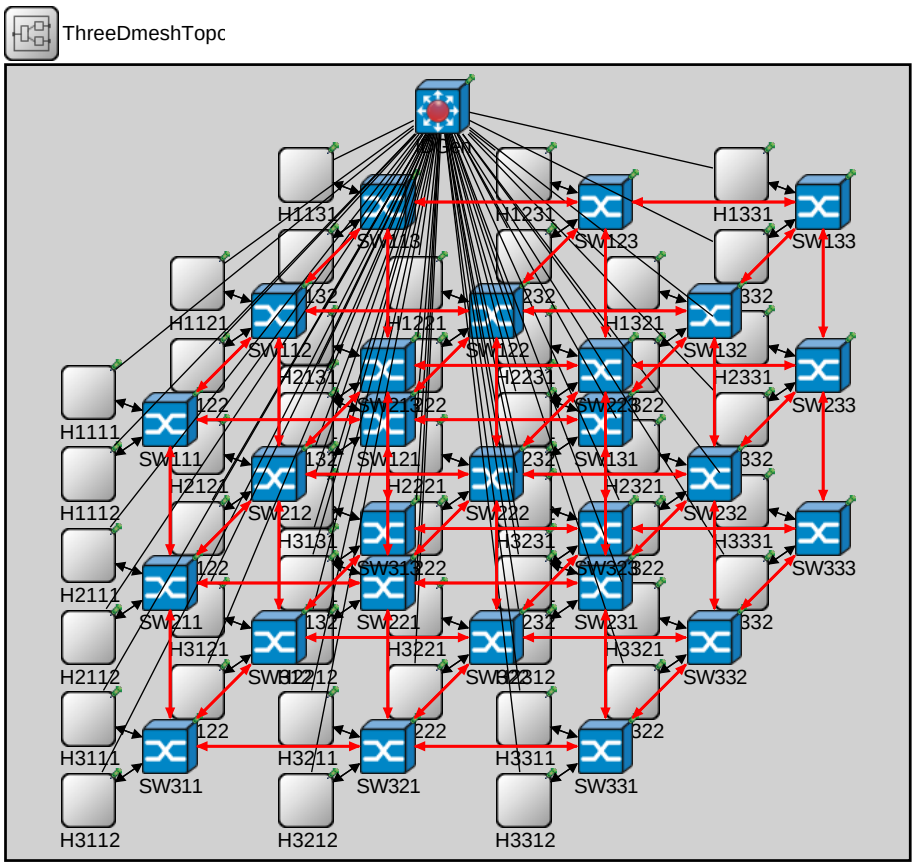

Figure 3.6: 3D Mesh Topology in OMNeT++ Environment.

3.5.4 Script: *2DMeshRouting.py*

This script has been developed to generate forwarding databases for any *mxm* 2D mesh topology based on DOR routing algorithm. The FDBs file content generated by the script for a 3x3 2D mesh network topology having two HCAs at each switch shown in Figure 3.4 above is as shown in Table 3.4 below. The file consists of index vectors of 9 switches in the left column and the output ports used by each switch to forward packets to 18 HCAs. The first value *i.e. 255* at each switch in FDBs is to represent an HCA that doesn't exist. The source LID of the HCAs should be given in their naming order in the topology starting from 1. Each column corresponds to a destination LID and the number at each place corresponds to the output port to use at the switch corresponding to the row. Similar to the topology scripts, these two scripts also accept the number of switches that will be in one dimension of the mesh topology and the number of HCAs that will be attached to each switch.

Table 3.4: FDBs for a 3x3 2D Mesh Topology

2Dmesh.fdbs File

```
11: 255 0 1 3 3 3 3 2 2 3 3 3 3 2 2 3 3 3 3
12: 255 2 2 0 1 3 3 2 2 4 4 3 3 2 2 4 4 3 3
13: 255 2 2 2 2 0 1 2 2 2 2 3 3 2 2 2 2 3 3
21: 255 2 2 3 3 3 3 0 1 3 3 3 3 4 4 3 3 3 3
22: 255 2 2 4 4 3 3 2 2 0 1 3 3 2 2 5 5 3 3
23: 255 2 2 2 2 3 3 2 2 2 2 0 1 2 2 2 2 4 4
31: 255 2 2 3 3 3 3 2 2 3 3 3 3 0 1 3 3 3 3
32: 255 2 2 4 4 3 3 2 2 4 4 3 3 2 2 0 1 3 3
33: 255 2 2 2 2 3 3 2 2 2 2 3 3 2 2 2 2 0 1
```

3.5.5 Script: *3DMeshRouting.py*

Similar to the script developed for the 2D mesh forwarding table above, *3DMeshRouting.py* is a python script developed to generate the FDBs of 3D mesh topology. Since 3D mesh topology has a relatively complex interconnection than 2D, manual population of the forwarding database is unmanageable and more complex. Table 3.5 shows an excerpt from the FDBs of a 3x3x3 3D mesh topology generated by the script. Due to the large size of the content of this forwarding database, only the first and last 3 switches' forwarding table is displayed in the table below. However, the actual table entry has 27 switches in the left column and 54 entries at each switch. If the content of this small-sized topology is this much larger, one can imagine that what much complex will be the size of the larger topology.

Table 3.5: FDBs for a 3x3x3 3D Mesh Topology

3Dmesh.fdbs File

```
111: 255 0 1 4 4 4 4 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3
       3 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3
112: 255 4 4 0 1 5 5 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3
       3 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3
113: 255 4 4 4 4 0 1 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3
       3 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3
.....
.....
.....

331: 255 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
       3 2 2 2 2 2 2 2 2 2 2 2 2 0 1 4 4 4 4
332: 255 5 5 5 5 5 5 5 5 5 5 5 5 3 3 3 3 3 3 5 5 5 5 5 5 5 5 5 5 5 3 3 3 3 3 3
       3 5 5 5 5 5 5 5 5 5 5 5 5 4 4 0 1 2 2
333: 255 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
       3 2 2 2 2 2 2 2 2 2 2 2 2 4 4 4 4 0 1
```

3.6 Experiment Design

In this thesis, several number of simulations will be designed and conducted using five VMs that will be created in the ALTO cloud as we discussed in Section 3.2. Since OMNeT++ was designed in a way that simulations on different machines must generate the same results [47], it is possible to simultaneously run on different machines. This will give us the opportunity to have multiple simulation runs in parallel to conduct multiple experiments with in short period.

To achieve the goal of the thesis, the simulations will be conducted in *four* steps. First, in Section 3.6.1, we will simulate our simulation network topology model for smaller network topology to verify that everything in the network topology model is going well and to illustrate that the IB CC able to avoid the negative effects of congestion such as victim flow and unfair share of network bandwidth due to the parking lot problem [16]. In the second step, in Section 3.6.2, the simulation will be run for relatively larger topology to demonstrate that the IB CC is still effective with larger topology similar to the smaller once done before. In the third step, in Section 3.6.3, we will simulate for different traffic patterns to identify the traffic pattern where the given configuration for IB CC is not effective due to improper parameter value settings. In all the three set of experiments, each scenario will run two times by enabling and disabling the IB CC for the purpose of illustrating the negative effect of congestion on the performance of the network and how this is avoided using IB CC. Finally, the bulk of the simulation runs will be made in Section 3.6.4 to find the proper IB CC parameters for the identified traffic pattern in the third step. After completion of this step, the best results will be recommended for the IB CC setting of such networks.

3.6.1 Experiment Set A: Simple Base Experiments

In this experiment, two simulations (one by disabling the IB CC and another by enabling the IB CC) will be conducted to verify that everything such as the routing and the topology designed in our network topology model is working properly. This can be demonstrated by the effectiveness of the IB CC to avoid the negative effects of congestion.

Topology and Traffic Flows

A 3x3 mesh topology which has two HCAs at each switch (9 switches and 18 HCAs) will be used for both experiments as shown in Figure 3.7. Even though only used HCAs are shown in the figure to avoid complexity in the drawing, it shouldn't be forgotten that all the switches are attached to two HCAs, this holds true for all the topologies used in all the experiments. The naming, number of ports, interconnection between nodes (switches and HCAs) will be as explained in Subsection 3.3.1 above since it will be produced using our script that will be developed for this purpose. A simple 3x3 topology is chosen because it has all the kinds of switches that exist

in any large 2D mesh topology. After the expected results are obtained from this simulation, we can further continue our simulation for larger topologies since the results will help us to verify that the simulation model is properly designed and configured.

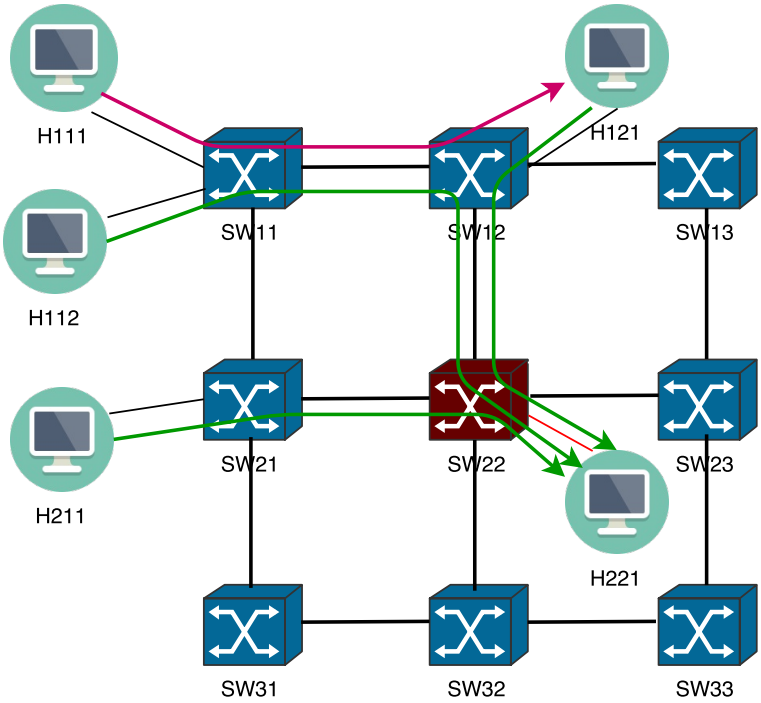

Figure 3.7: Experiment A Topology.

As depicted in Figure 3.7, *three* traffic flows as indicated in 'green' arrows from contributors *H112, H211* and *H121* is destined towards *H221*. The node *H112* generates and sends traffic in its full capacity to *H221* when the simulation started. The traffic from *H112* flows through *SW11* to *SW12* then to *SW22* and finally reaches to *H221*. *H211* and *H121* also send traffic in their full capacity to the same destination, *H221*. Finally, *H111* starts sending traffic to a different destination *H121* through *SW11* and then via *SW12* as indicated in 'pink' arrow in the figure. As a result, *H121* will be both a contributor and a sink HCA and *H221* will never generate a traffic. All the HCAs sent their traffic to a given static destination using the destination LID.

Experiment A: IB CC Disabled

In this experiment, everything will be kept similar to the experiment below except that the IB CC is going to be disabled. The aim of the experiment is to assert the performance gains of the IB CC that will be used.

Expected Results

When the simulation starts, *H112* uses the network resources alone with 100%. Next, when *H211* starts sending traffic, the two HCAs equally share

the network resources with 50% each. When the third HCA, *H121*, starts sending, it only shares from 50% of the resources *H112* is using since they are sharing the same link and port. So, *H121* and *H112* will have 25% share for each, due to the parking lot problem. *H211* will keep it 50% share. Now, *H111* is also affected by the congestion although it doesn't share the bottleneck link and doesn't contribute for the congestion to occur. Thus, *H111* will be a *victim* HCA which results in a victim flow. So, *H111* can only send with equal share of the traffic with *H112*, i.e. 25%.

Experiment A: IB CC Enabled

This experiment can be considered as a verification test of our simulation model and the effectiveness of the IB CC. The main purpose of the experiment is to check the IB CC, the topology, the routing table, the port-wise interconnection and others are properly configured and working as intended before going into experiments involving lager topologies.

IB CC Parameter Values

We intentionally make the the traffic pattern to look like as 2x2 mesh topology as shown in Figure 3.7 because we planned to base the initial IB CC parameter values used for our experiment to be taken from a previous study [29]. These parameter values are tested and referred as the best parameter values for a 2x2 mesh in the study and can be used to verify the correctness of our network topology model. These IB CC parameter values that will be used in this experiment are shown in Table 3.6 below:

Parameters	**Values**
Threshold	2
Marking_Rate	1
Packet_Size	1
CCTI_Timer	20
CCTI_Limit	128
CCTI_Min	0
CCTI_Increase	1

Table 3.6: Initial IB CC Parameter Values

Expected Results

At this time, when the simulation starts, *H112* uses the network resources alone with 100%. Next, when *H211* starts sending traffic, the two HCAs equally share the network resources with 50% each. When *H121* starts sending the traffic, three of them equally shares the network resources with 33.3%. Since all of the are sending to the same destination at the same port, the link between *SW22* and *H221* will be a bottleneck link. Thus, *SW22* will be the hotspot switch which will detect congestion when its buffer starts to fill up and reaches a certain threshold. Finally, when *H111* starts sending, the traffic will reach to the destination unaffected by the congestion since

it doesn't share the bottleneck link. As a result, *H111* will able send traffic with its full capacity, 100%.

3.6.2 Experiment Set B: Static Network Traffic Pattern

This set of experiments can be considered as the first large 2D mesh topology experiments in our network topology model using static hotspot traffic pattern. The main purpose of these set of experiments is to understand the benefits of IB CC in a large 2D mesh topology. In addition to this, the experiment will be used to proof the forwarding tables created using the DOR routing algorithm is implemented correctly. These set of experiments will provide us in what situations will a node will be a victim by placing the expected victim node in different positions in the topology. Since exploring the proper parameters of IB CC for a given topology requires a number of step by step experiments, this will be the first step to approach the problem statement.

In this section, we will have *four* experiments. In all the experiments, we will have *five* contributors destined to the same destination, one victim node destined to a different destination and one hotspot switch. All other nodes in the network will not generate any traffic. Among these four experiments, in two of them the IB CC will be enabled, and will be distinguished each other by varying the position of the victim node. The other two experiments will have the same configuration with the first two experiments but IB CC will be disabled in this case.

Experiment B1: IB CC Disabled

In this subsection, two experiments, by enabling and disabling the IB CC, will be done. This experiment is similar to the experiment below except the IB CC will be disabled to check that whether *H431* is a victim node or not.

Experiment B1: IB CC Enabled

The number of nodes contributing traffic, the victim node, the hotspot switch, the bottleneck link, the sink nodes and generally the topology and the traffic flow will be explained in the next subsection.

The expected result for these two experiments above are:

- The DOR algorithm will work properly, and the node *H431* uses the x-direction, and it will not be a victim node.
- Consequently, whether the IB CC is enabled or disabled it doesn't have an effect for *H431*. So, the performance of this node will be the same for both experiments.

Topology and Traffic Flows

The topology chosen for this experiment is a 5x5 mesh in which all the switches are attached to two HCAs, i.e. there will be 25 switches and 50

HCAs in the network topology as shown in Figure 3.8. The nodes *H111*, *H151*, *H511*, *H541* and *H551* are the contributors of congestion which sends traffic to the destination node *H552* as indicated in 'green' arrows. As a result, the link between *SW55* and *H552* will be the bottleneck link and *SW55* will be the hotspot in which congestion starts to occur.

The node *H431* sends traffic to the destination *H542*. If the forwarding table is correctly implemented based on DOR algorithm, it will not be a victim node since it sends its traffic first in the x-direction as indicated in 'pink' arrow. This is because if the node uses this direction, it doesn't share any link and port with other nodes. However, if the DOR algorithm is not properly working, *H431* may send its traffic first in the y-direction as indicated in a 'red' dashed-arrow. In this case, the node will share the link between *SW53* and *SW54* and will be a victim node although it doesn't share the bottleneck link.

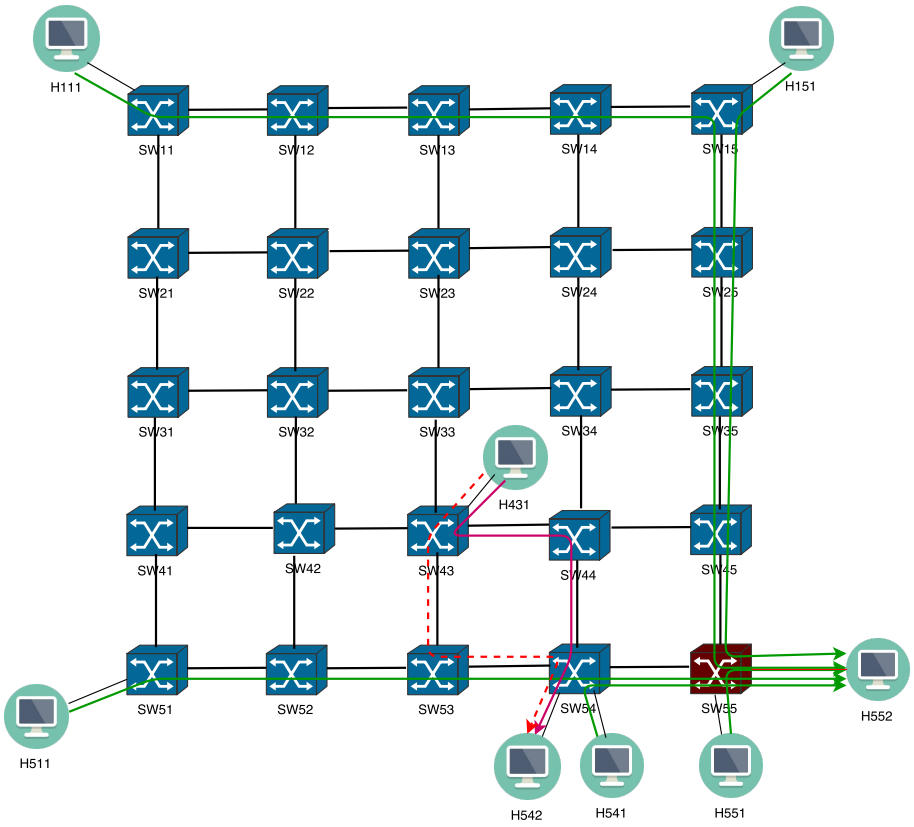

Figure 3.8: Experiment B1 Topology.

Experiment B2: IB CC Disabled

As in subsection above, there will be two experiments in this subsection as well. These experiments are expected to proof the real benefits of using IB CC in large 2D mesh topology. Unlike experiments in the above subsection, one node will be place in the appropriate position in the topology to be a victim node. The aim of this part is to crosscheck that what will happen in

the network traffic if the congestion control is not used.

Experiment B2: IB CC Enabled

In both experiments, the start up times of the hosts will be similar to the one used in the above subsections except that node *H512* will start instead of *H431*. Thereafter, we will observe that how this node will be benefited from the IB CC when it is enabling. When the IB CC is enabled, the following results will be expected:

- The overall network performance increases.
- There will be fair share of the network resources.
- There won't be victim node.

Topology and Traffic Flow

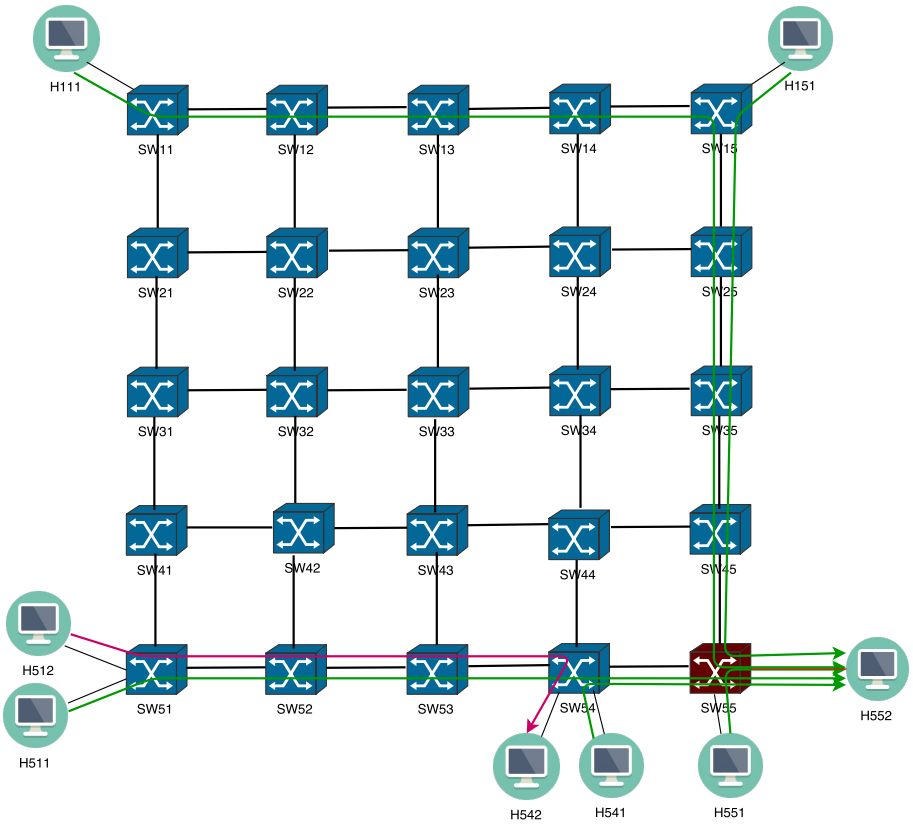

Figure 3.9: Experiment B2 Topology.

The topology, the contributor nodes, the sink nodes and the hotspot switch will be kept the same as those used in the above subsection. The only difference is *H431* will be replaced by *H512*, as shown in Figure 3.9. The new node, *H512*, has no any alternative path to reach its destination, *H542* rather than sharing the link with *H511*. Due to this, *H512* will be a victim

node and affected by the congestion when it is occurred in the network due to the bottleneck link between *SW55* and *H552* although this node doesn't request this link.

3.6.3 Experiment Set C: Exploring Different Traffic Patterns

In the previous experiments a fairly simplistic traffic patterns in which each traffic source node sends to only a single destination node is used. This limited number of communication per source node may not reflect the traffic patterns in many InfiniBand network applications. Thus, more general experiments that varies the number of connections per source node was used in this subsection. The main aim of the experiments in this subsection is to find the traffic pattern to which the InfiniBand Congestion Control is not efficient enough due to the IB CC parameters used are not the proper parameters for the given traffic pattern. In all the scenarios, two simulation runs will be made by enabling and disabling the IB CC. This will help us to investigate in which scenario the IB CC is more inefficient and then needs a parameter value change. After the traffic pattern is identified, it will be chosen for further simulations (Section 3.6.4) by varying the IB CC parameter values until the best parameter values are obtained for the chosen traffic pattern.

Traffic Patterns

The traffic pattern of the packets has a great impact on the performance of the network [21, 38]. As a result, the performance of interconnection networks has been evaluated with different types of traffic patterns. Among these, *uniform random* and *hotspot* traffic patterns are the commonly used once.

- ***Uniform Random Traffic***: Each node sends packets to any of the other destination nodes with an equal probability. The destination nodes will be chosen randomly using a uniform probability distribution function.
- ***Hotspot Traffic***: Each node sends packets with an equal probability except for a specific node known as *Hotspot* which receives packets with a greater probability.

In our simulation, we will use either of the two traffic patterns or a combination of them. For instance, the traffic pattern used in Subsection 3.6.1 and 3.6.2 is a hotspot traffic pattern. However, in this subsection, we will use a combination of uniform random and hotspot traffic pattens.

For the next three experiments (Expriment C1 - C3), a similar topology, static traffic contributors, victim node, destination nodes, bottleneck link and hotspot switch shown in Figure 3.9 will be used. However, at every simulations in each subsection, other specified percentage of the nodes in the network will send traffic to random destination. Consequently, the

traffic pattern will be changed by varying the number of uniform random sending and receiving nodes in the network for each simulation.

For the last subsection experiment (Experiment C4), a different number and type of static traffic contributors, victim nodes, destination nodes, bottleneck links and hotspot switches shown in Figure 3.10 will be used. In this case, three hotspots will be used and the traffic pattern will be completely changed. The following subsections present the details of the traffic pattern of each simulation.

Experiment C1: 100%-100% Uniform Random Traffic Pattern

As we discussed above, the topology and traffic flow used in this experiment will be the same as the one used in Section 3.6.2 (ExperimentB1). In this scenario, in addition to the hotspot traffic pattern, the other 100% of the nodes will send a uniform random traffic for the rest 100% of the nodes. As a result, it is expected that the congestion will be occurred and the efficiency of the IB CC will be challenged with the given parameter.

Experiment C2: 50%-100% Uniform Random Traffic Pattern

In this scenario, only 50% of the nodes will send uniform random traffic to the rest 100% of the nodes in the network. The network congestion in this case will be a little bit better since the number of nodes consuming the packets are larger than the traffic contributors.

Experiment C3: 100%-50% Uniform Random Traffic Pattern

In this scenario, 100% of the nodes in the network will send for 50% of the nodes in the network. Therefore, the probability of the congestion to occur will be relatively higher than the above two experiments. Although this scenario has a similar number of nodes that generate traffic with *Experiment C1*, only 50% of the nodes consumes the traffic. Therefore, this scenario will be more congested.

Experiment C4: Uniform Random Traffic Pattern with Three Hotspots

In this scenario, three switches will be selected to be hotspot switches. Then, only 30% of the nodes will be allowed to send static traffic to the hotspots so that 10% of the total nodes in the network to send per hotspot. The remaining nodes in the network will send uniform random traffic for 100% of the nodes in the network. This scenario is expected to produce the most congested network in which the IB CC will even perform to the worse, and will be the case for the largest potential for improvement. If the results are obtained as expected, this will probably be the traffic pattern used for the experiment in Section 3.6.4.

As shown in Figure 3.10, *H551, H541, H211, H141* and *H251* are contributors which will send traffic to the destination node, *H241*, as indicated in 'green' arrows. Thus, the link between *SW24* and *H241* will be

a bottleneck link for this traffic flow. The switch *SW24* will be a hotspot. Another node, *H212*, will send traffic to the destination node, *H231*, as represented in 'pink' arrow. This node will be a victim node. On the other hand, the contributor nodes, *H111, H121, H511, H451* and *H441* will send traffic to another destination node, *H421*, as indicated in 'green' arrows so that the link between *SW42* and *H421* will be a bottleneck link and the switch, *SW42*, will be a hotspot. The node *H131* is a victim node which will send traffic to another destination node, *H321* as represented in 'pink' arrow. Furthermore, the contributors,*H151, H142, H411, H532* and *H422*, will send traffic to the destination node, *H531*, as indicated in 'green' arrows. Consequently, the link between *SW53* and *H531* will be a bottleneck link and the switch, *SW53*, will be a hotspot. Another node, *H132*, will send traffic to to a different destination node, *H331*, as indicated in 'pink' arrow. As a result, *H132* will be a victim node.

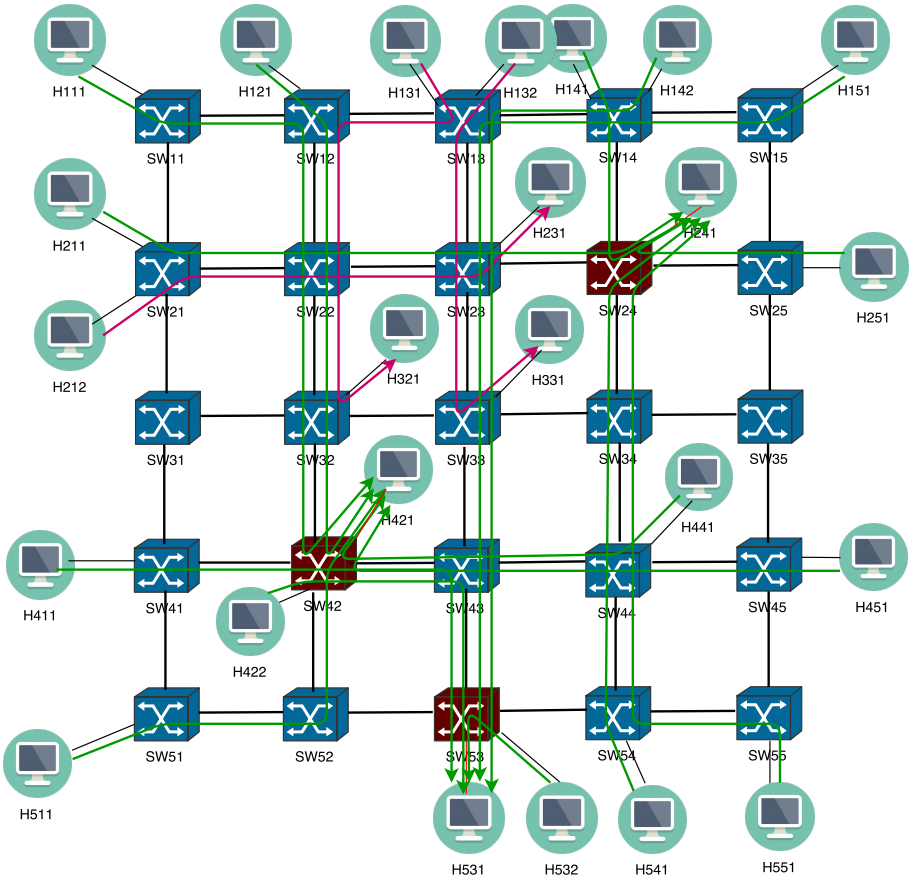

Figure 3.10: Topology for Uniform Random Traffic Pattern with Three Hotspots.

3.6.4 Experiment Set D: Finding Proper IB CC Parameter Values

The experiments in this subsection will consist of the majority of the simulations. The main focus of this thesis will be this subsection since the

target is to find the proper parameters for a mesh topology that ensure the maximum efficiency utilization of the IB CC for a given topology.

Finding the proper parameters will be a challenging and time consuming task. As we discussed in Section 2.3.2, there are several IB CC parameters which will make the exploration of all the parameter spaces very difficult in short period of time. Therefore, we have to choose and limit ourselves for some specific IB CC parameters in this work. Two most important parameters, one from the switch and another form the channel adapter, will be chosen and studied for this thesis. So, from the switch, *Marking_Rate* and from the channel adapter *CCTI_Timer* will be chosen in our case. Using this parameters several number of simulations will be conducted to determine which parameter values will be best suited for this network. At each simulation, the value of one parameter, say the value of *CCTI_Timer*, will remain as it is while the other one, say the value of *Marking_Rate*, is changed. Then, the vise versa will be done. These two values are chosen because they are the most relevant parameters [15] that significantly determine the performance of the IB CC.

3.7 Data Collection and Plotting

After all the simulations are successfully accomplished, OMNeT++ will produce large files (in GB) that need further processing to extract relevant values for better analysis. This Section presents the techniques and tools used to collect, process and plot data from OMNeT++ result files.

3.7.1 Data Collection

If OMNeT++ is up and running, and our network simulation model is properly designed and configured, gathering data from the network simulation is not a difficult task. OMNeT++ is able to gather data and save simulation results into vector (**.vec*) and scalar (**.sca*) files. Vector files include data values as a function of time, while scalar files include aggregate values at the end of the simulation. To enable our simulation to record data values and events, it is a matter of including a single extra line in the simulation's configuration **.ini* file as follows:

```
Record vector and scalar files
record-eventlog = true
```

3.7.2 Data Processing

One of the challenging part while using OMNeT++ is processing these quite large vector and scalar result files and extracting the relevant values from them. There has been no a single straightforward and efficient way to read, process and plot the contents of OMNeT++ result files until now [37]. However, since the output file format is simple and straightforward to be used in scripting languages, the OMNeT++ community writes their own

result processing scripts in *Perl, Python, shell scripting* and other languages. In addition, several open-source tools and libraries such as *grep, awk* and *scavetool* are available for loading and basic processing OMNeT++ result files. *scavetool* [46] is suitable for filtering and basic processing of result files but it has no graphics capabilities.

In our work, we will use a *python* [39] programming language together with *grep* [28], a Linux command-line utility, to process and extract relevant values from these quite large OMNeT++ result vector files. Python is chosen because it is a powerful programming language and we have a long time experience in this programming language. After the basic filtering and processing of OMNeT++ result files is made using python, the results can be saved in various formats such as *CSV, Matlab, Octave*, etc which are suitable to be plotted with tools like *R, Matlabplotlib, Excel, gnuplot2*, and so on.

3.7.3 Plotting

The OMNeT++ IDE [48] provides a result analysis tool, *Scave*, to support filtering, processing and visualizing simulation results saved into vector and scalar files. However, unfortunately, the tool is quite complex with little documentation yet. In addition, the IDE's analysis tool currently doesn't support the generation of publication-quality plots. Due to this, currently, there is an effort to develop *OMNeT++ R extension* package, called *omnetpp* [45]. The *omnetpp* package is built upon the scavetool's C++ library and R scripting. The omnetpp package was written to facilitate evaluating OMNeT++ simulation results in R. The package is designed to support the reading, processing and easy plotting of OMNeT++ result files, and adding to the vast arsenal of statistical computing and graphical capabilities of R. The first incomplete but already useful version is available for download and use [45]. However, since this package is a new release and is not a full-fledged tool as well as has no enough documentation about it, we are unable to use the package.

Consequently, we'll look for an easy and powerful solution to produce a simple vector file plots which is already processed by python scrips. So, for a better plotting, we are planning to use R with *ggplot2* [49] which can generate publication-quality plots using the outputs of the python scripts. R [50, 43] is a powerful language and open-source software environment designed for statistical analysis and excellent graphics. R is already being used by many users in the OMNeT++ community despite the fact that there has been no straightforward way to read result files into it.

Chapter 4

Results and Analysis

This chapter presents the results of the experiments described and designed in Chapter 3. It also provides a comprehensive analysis of the results obtained in all the scenarios described in Section 3.6. These experiment results are the main focuses of this thesis, thus, a number of simulation results will be presented and analysed in the next five sections. The goal of the analysis is to verify the efficiency of the IB CC to avoid the negative effects of congestion and to explore the proper IB CC parameter values that allow to attain the full capability of the IB CC.

4.1 Simple Base Experiments

Figure 4.1 and 4.2 show the simulation results of the experiments designed in Section 3.6.1 (*Experiment Set A*) with the topology and traffic flow in Figure 3.7 when the IB CC was disabled and enabled, respectively. The results are largely the same as the expected results.

Figure 4.1 shows the result of the simulation when the IB CC was disabled (*Experiment A: IB CC Disabled*). All the parameters in the configuration, the traffic generation start up time of the nodes and the destination nodes were similar to the one below except that the IB CC was disable in this case.

At the beginning, *H112* started sending traffic at its full capacity and used the channel alone with 100% as indicted in 'red' line. When *H211* started sending traffic after *0.1sec* as indicated in 'green' line to the same destination, the 'red' line dropped half way because it was sharing the link with 50% with this node. When a third node, *H121*, started sending traffic as indicated in 'pink' line, the 'red' line dropped an other half way and shared its 50% share equally with 25% each because the two nodes shared the same port and treated as a single flow by the switch. However, node *H211* still kept its 50% share. At this time, the destination node *H221*, received traffic 50% from *H211*, 25% from *H112* and 25% from *H121* and kept receiving this amount of traffic as indicated in 'black' line.

Finally, when *H111* started sending traffic after *0.4sec* as indicated in 'blue' line to a different destination *H121*, *SW11* is a fair switch it alternatively chose packets from *H111* and *H112*. Since the share of *H112*

was only 25%, *H111* was also restricted to send only 25%. Although this node didn't share the bottleneck link, it became a victim node due to the absence of the congestion control. The packets received by the destination node *H121* was also limited to 25% like *H111* as indicated in 'cyan' line although the link between *SW12* and *H121* was free.

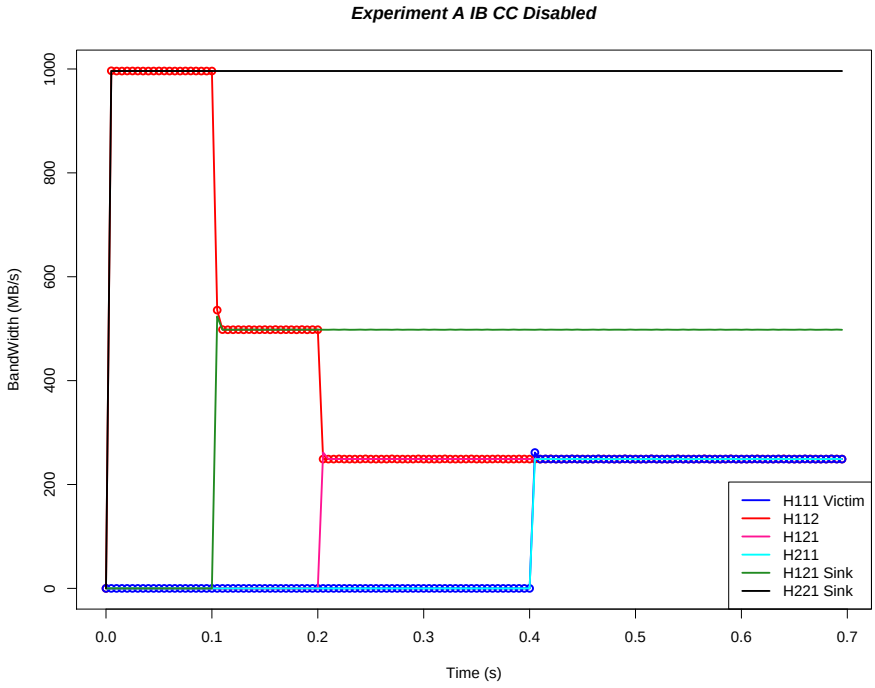

Figure 4.1: 3x3 2D Mesh Topology Result When IB CC Disabled.

Figure 4.2 shows the result of the simulation when the IB CC was enabled (*Experiment A: IB CC enabled*). As shown in the figure, *H112* started generating and sending traffic in its full capacity to the destination *H221* with a bandwidth of about 1000MBps right after the simulation was started. This node utilize the channel alone until the second node was started sending traffic as represented in 'red' graph. After *0.1sec*, *H211* started sending traffic which was destined to the same destination as indicated in a 'green' line in the graph. After this time, the 'red' graph dropped half way down because the two nodes should share the network equally with 50% each. Then after *0.2sec*, a third node, *H121* started sending traffic again to the same destination as represented in 'pink' graph. Since *SW22* was a fair switch and the IB CC is on, the three contributors shared the channel equally with 33.3% each. All the traffic sent by the contributors were received by the sink node, *H221*. As a result, this destination node kept receiving the same amount of traffic as indicated in 'black' graph, which is the sum of the traffic generated by the three contributors.

Finally, *H111* started sending traffic after *0.4sec* as represented in 'blue' graph to a different destination *H121*. Since this node didn't share the bottleneck link, it was able to send traffic in its full capacity. All the

traffic generated by this node arrived at its destination. The 'cyan' graph which overlapped with the 'blue' graph indicted that all the traffic generated by *H111* was received by *H121*. Even though, the simulation continued till *T = 0.7sec*, the graphs didn't changed and showed the same behaviour. Consequently, the IB CC avoided HOL blocking and there was fair share of resource sharing that eliminate the parking lot problem although congestion was occurred in the network.

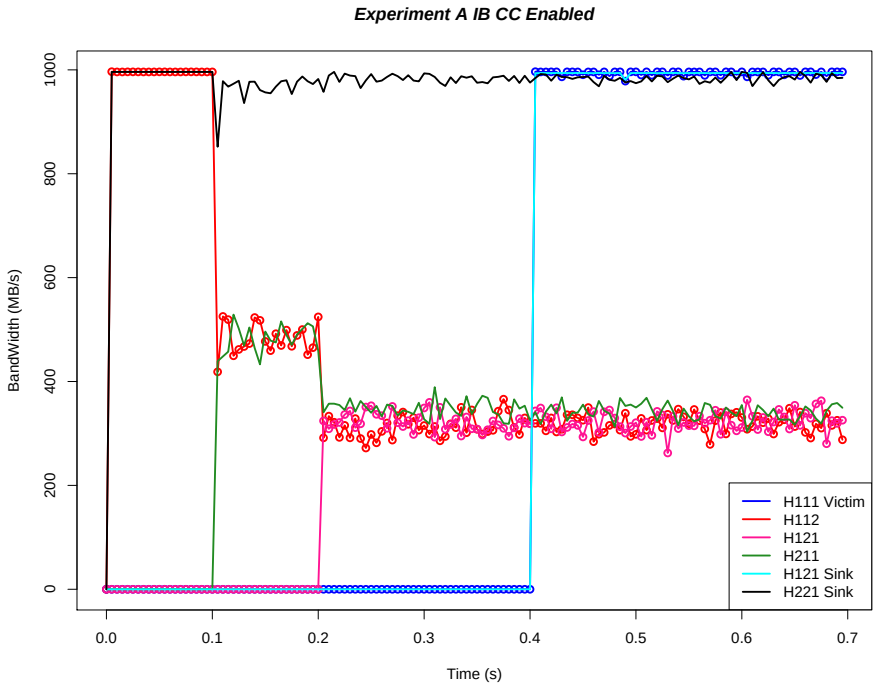

Figure 4.2: 3x3 2D Mesh Topology Result When IB CC Enabled.

4.2 Static Network Traffic Pattern

This section presents the simulation results of the experiments described and designed in 3.6.2 (*Experiment Set B*). This experiment was conducted with the same IB CC parameter configuration and simulation time with the above experiment. The first two figures, Figure 4.3 and Figure 4.4 are the results of the experiment designed in 3.6.2 (*Experiment B1*) with the topology and traffic flow in Figure 3.8 when the IB CC was disabled and enabled, respectively.

The time at which each node started sending traffic is summarized in Table 4.1. This traffic generation start up time was used for all the experiments in this section. The simulation continued till *T=0.7sec*.

HCAs	Start times (s)
H111	0.0
H511	0.1
H551	0.2
H151	0.3
H541	0.4
H431	0.5

Table 4.1: HCAs starts sending traffic.

When the IB CC is disabled, the resulted graph looks like as shown in Figure 4.3 which is the same as the expected result. The graph shows that there is a *parking lot problem*.

The first three nodes, *H111, H511* and *H551* as indicated in 'blue', 'pink' and 'black' lines, respectively, shared the bandwidth equally with 33.3% each since they were using different ports at the hotspot switch *SW55*. However, when the fourth node *H151* started as indicated in 'red' line, it shared only the share of *H111* because the two nodes used the same port at the hotspot switch and treated as a single flow. When *H541* started as indicated in 'cyan' line, it shared the bandwidth with *H511*. This is because the two nodes shared the same port at the hotspot switch and treated as a single flow. The total throughput at the destination node *H552* is represented in 'dark' green.

The node *H431* was not a victim node as indicated in 'gray' line. This is because in DOR algorithm, the flow used the x-direction first. If it sent first in the x-direction, the flow didn't share any resource with other nodes and wouldn't be a victim. However, if it had been using first the y-direction, it would be a victim flow. Which means the DOR algorithm was implemented properly. The 'yellowgreen' line represents the traffic received by the destination node *H542* which is equal to the sending node *H431*.

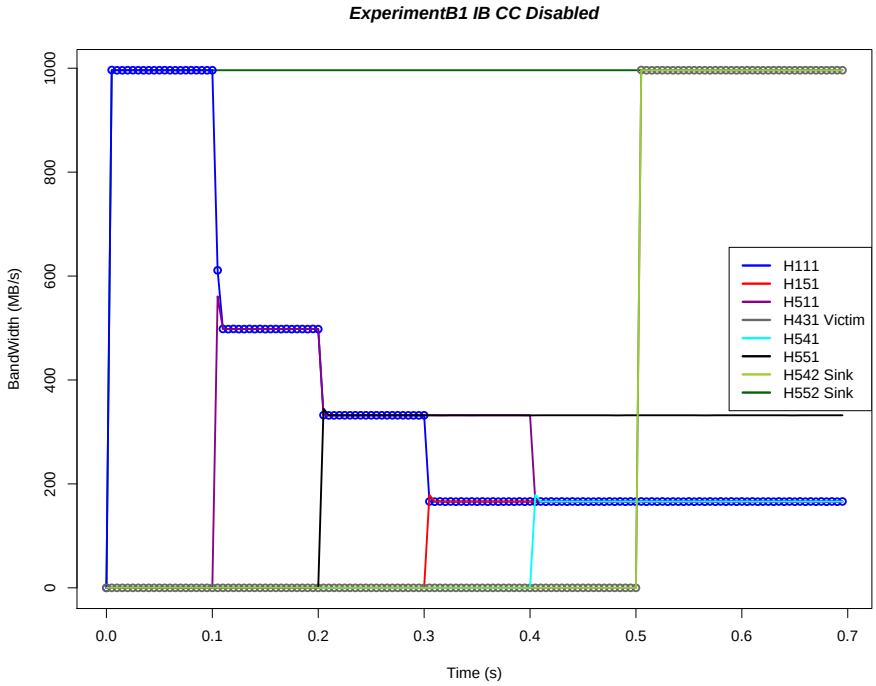

Figure 4.3: 5x5 2D Mesh Topology Result When IB CC Disabled.

The results obtained in this scenario are largely similar to the above experiment and it reflects the expected results. Even though, there was no victim flow in this scenario, the IB CC solved the *parking lot problem*. As the graphs in Figure 4.4 show, there was fair share of the bandwidth among the nodes.

At the beginning, *H111* started sending traffic in its full capacity to the destination *H552* at a bandwidth of about 1000MBps immediately after the simulation was started as indicated in 'blue' line. This node kept using this bandwidth until the second node, *H511*, was started sending traffic as indicated in 'pink' line to the same destination. When *H511* started sending traffic, the two nodes shared the bandwidth equally with 50% each. After *H551* started, as represented in 'black' line, the three nodes shared the bandwidth equally with 33.3% each. Then the fourth node *H151* started sending traffic as indicated in 'red' line and shared the bandwidth again with 25% each. After the fifth node *H541* started sending again to the same destination as indicated in 'cyan' line, the bandwidth was shared by the three nodes with 20% each. The 'darkgreen' line is the bandwidth throughput at the destination node, *H552*, which keeps receiving equal amount of packets from the beginning to the end of the simulation. Finally, after *0.5sec*, the sixth node *H431* started sending traffic to a different destination *H542*, it able to use the bandwidth with its full capacity all the time since it didn't share any resource with other nodes, as indicated in 'gray' line. The 'yellowgreen' lines represent the traffic received by the destination node *H542* which is the equal to the sending node.

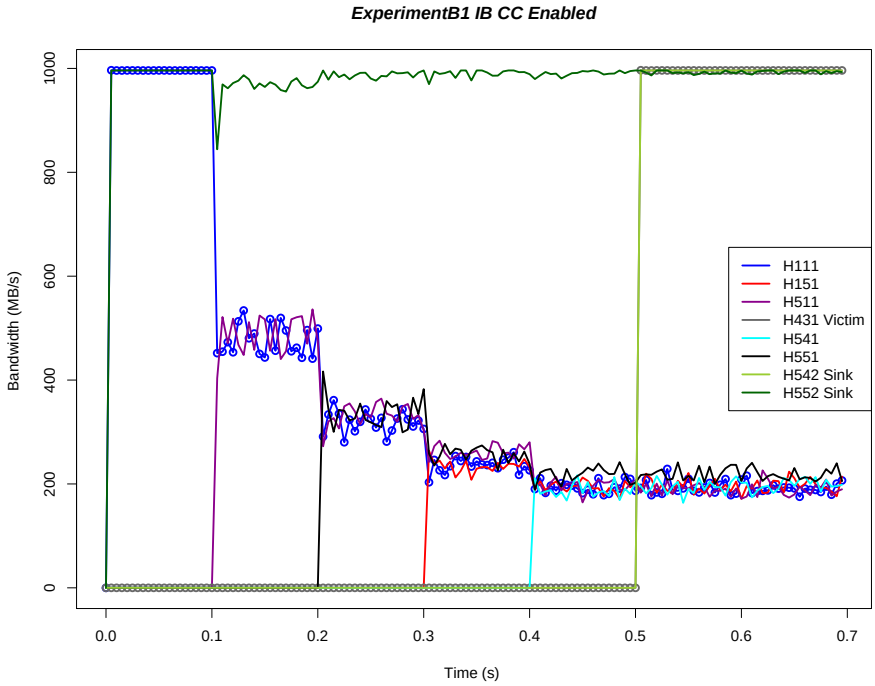

Figure 4.4: 5x5 2D Mesh Topology Result When IB CC Enabled.

The next two figures, Figure 4.5 and Figure 4.6 are the results of the experiment designed in 3.6.2 (*Experiment B2*) with the topology and traffic flow in Figure 3.9 when the IB CC was disabled and enabled, respectively.

Even though, the sixth node *H431*, which was sending to a different destination *H542*, was replaced by *H512* which shared links with another node *H511*, the graphs in Figure 4.4 are very similar to the graphs in Figure 4.6. This is due to the fact that the IB CC avoided the problem of victim flow due to congestion. When the IB CC was disabled the graphs shown in Figure 4.3 and the graphs shown in Figure 4.5 look like identical except one flow, the *victim flow*. This is due to *H512* was a victim node and should share the same bandwidth as the node *H511* as indicated in 'gray' line. The throughput at the receiving node *H542* was also dropped as represented in 'yellowgreen' line in Figure 4.5.

By enabling the IB CC, as shown in Figure 4.6, both *HOL blocking* and *parking lot problem* are solved.

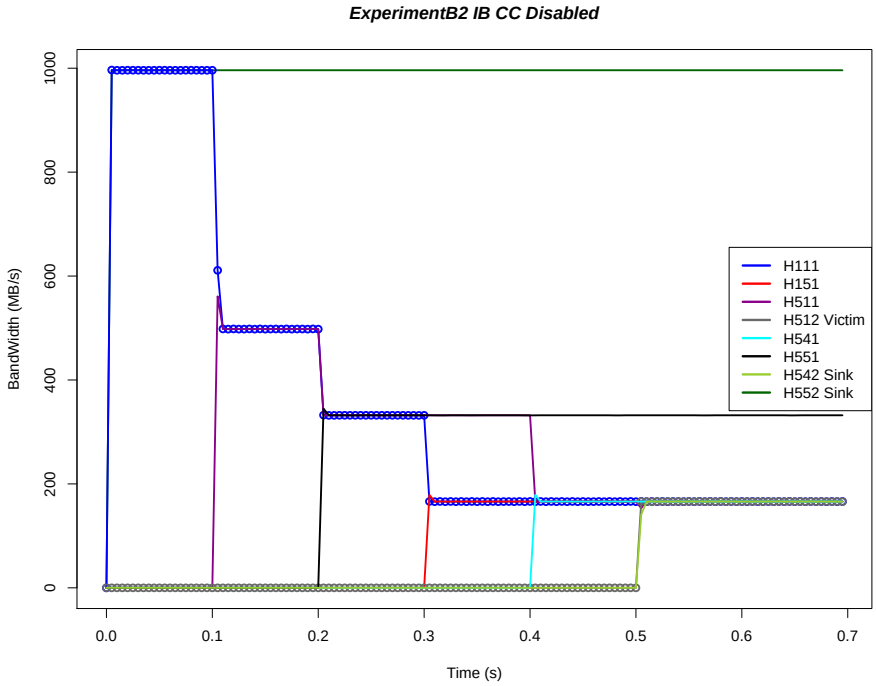

Figure 4.5: 5x5 2D Mesh Topology Result When IB CC Disabled.

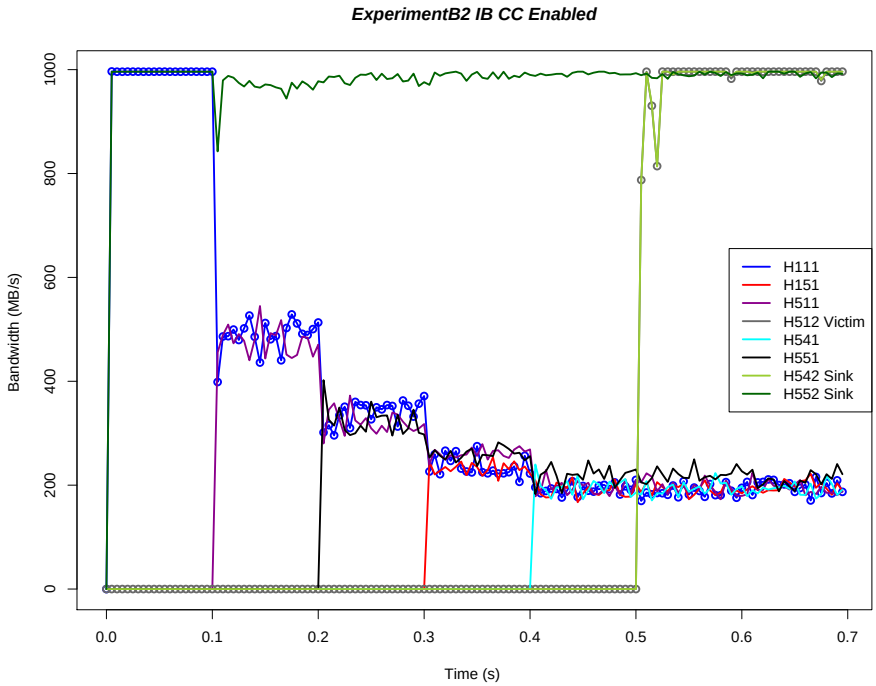

Figure 4.6: 5x5 2D Mesh Topology Result When IB CC Enabled.

4.3 Exploring Different Traffic Patterns

The experiment results of different traffic patterns described in Section 3.6.3 (*Experiment set C*) will be presented and analysed in this Section. In all the experiments, the IB CC parameters values, the topology and other configurations were similar to the previous experiments except the traffic pattern.

4.3.1 100%-100% Uniform Random Traffic Pattern

As we described in Section 3.6.3 (*Experiment C1*), the traffic patterns used in this scenario is both hotspot and uniform random traffic patterns. In this scenario, 100% of the nodes sent for a randomly chosen node out of 100% of the nodes in the network, in addition to the contributors and the victim nodes. The first two graphs shown in Figure 4.7 and Figure 4.8 are the results of the particular nodes sending traffic to the hotspot when the IB CC was disabled and enabled, respectively.

The y-axis in the left side is the scale for the bandwidth used by the contributors in both figures. Each node unable to use more than 250MBps bandwidth. Then, the performances of all nodes were dropped radically due to all nodes in the network generated and sent traffic to random destinations which resulted congestion in the network. With the deterministic DOR routing in 2D mesh topology and traffic flow such as the one used in Figure 3.9 for this experiment, the right side links will be the most congested links. It is expected to be the same for the links to the left for the traffic going in the opposite direction.

Particularly, the graphs of the nodes at the far corner in the topology from the destination were at the bottom because they have to share the bandwidth from the source to the destination with many of the nodes along their path and uses the most congested link. Due to this *H111* was at the bottom as indicated in 'blue' line, on the other hand, *H551* performed relatively better as indicated in 'black' since it was near to the destination node and didn't share any congested link.

The y-axis in the right side is the scale for the throughput at the destination nodes *H552* and *H542*, represented in with 'darkgreen' and 'yellowgreen' lines in the graphs. The sink node *H552* received packets at its full capacity about 1000MBps since it received packet from several static sources and random sources. However, the node *H542* received packets only from one static source (*H512*) and random sources, so, it received packets less than *H552*.

The performance became the worst when the IB CC was disabled for the far corner nodes such as *H111* and *H151* as shown in Figure 4.7. The victim node *H512* had a better performance while the IB CC was enabled as shown in 'gray' line since it was sending to the less congested link but its performance suffered by congestion when the IB CC was disabled.

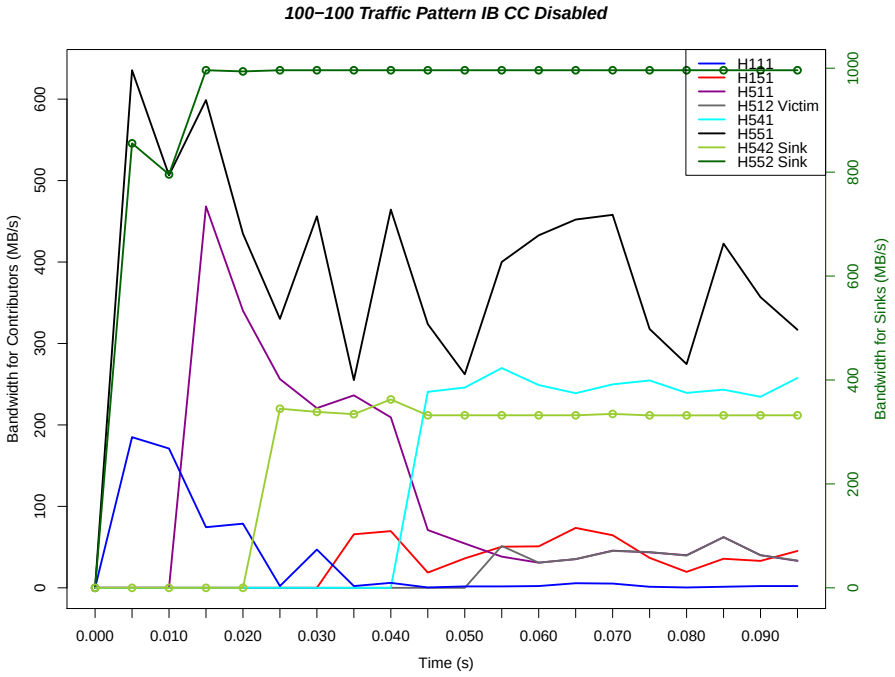

Figure 4.7: The performance of nodes sending to the hotspot in 100%-100% Hotspot Traffic Pattern when IB CC is Disabled.

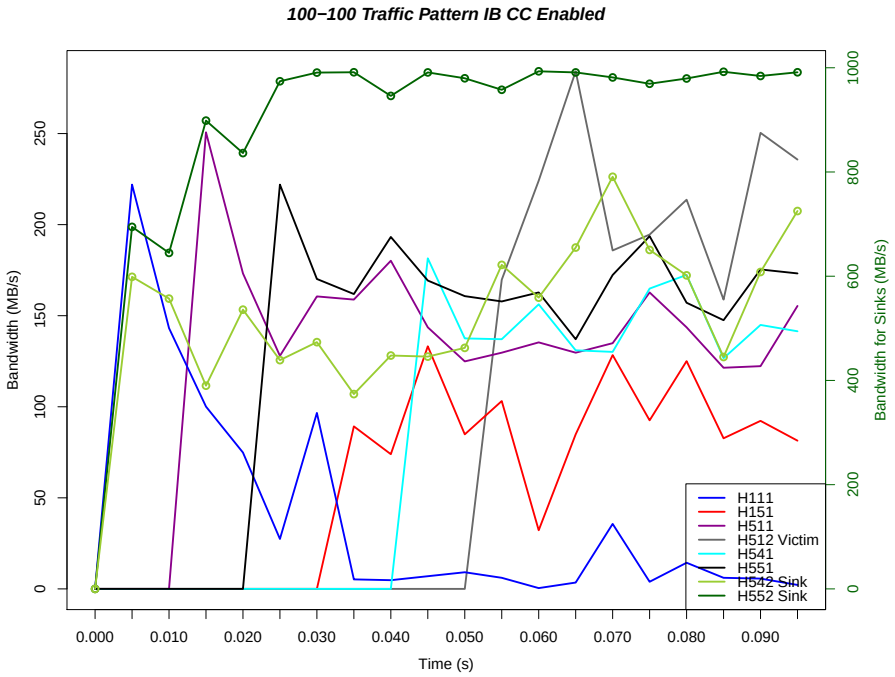

Figure 4.8: The performance of nodes sending to the hotspot in 100%-100% Hotspot Traffic Pattern when IB CC is Enabled.

In this section, we are interested in the overall performance of the network when the IB CC was disabled and enabled . Consequently, we depicted Figure 4.9 which shows the overall network throughput for the two cases. As shown in the figure, the performance was rapidly decreasing at the beginning due to the network traffic instability which leads to unacceptably low throughput, but later it started increasing. When the IB CC was enabled, the overall performance of the network was better as indicated in 'red' line in the graph and the average throughput was 25Gbps. However, the 'light-blue' graph in the figure was the overall throughput when the IB CC disabled and its average throughput was about 20Gbps at this time. This implied that the IB CC was still a better choice for such networks.

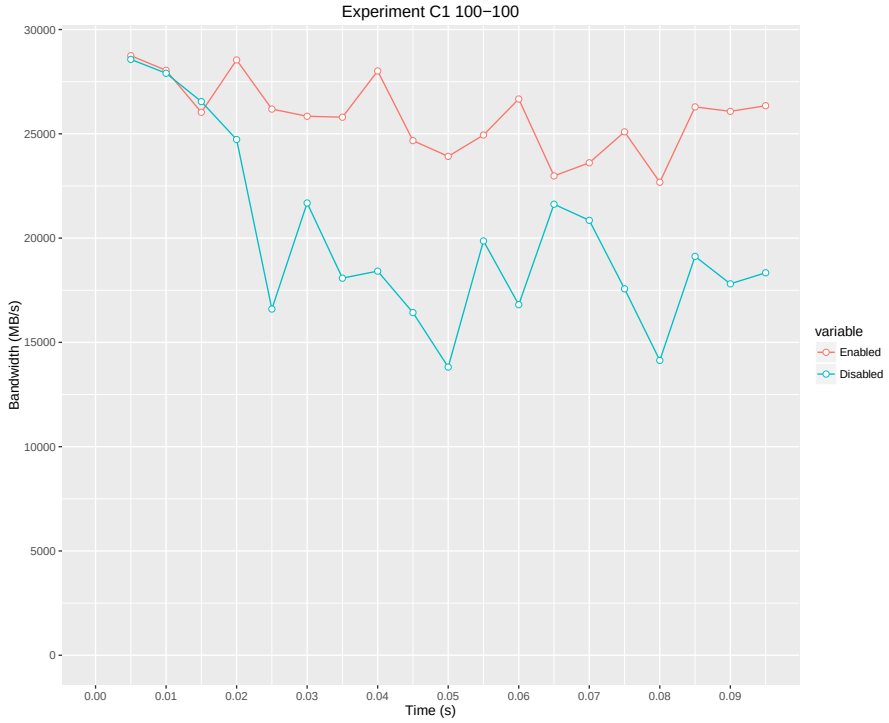

Figure 4.9: 100%-100% Uniform Random Traffic Pattern Overall Network Performance.

4.3.2 50%-100% Uniform Random Traffic Pattern

This section presents and analysis the results obtained for the scenario in which 50% of the nodes were sending for a randomly chosen node from 100% of the nodes in the network, in addition to the contributors and the victim nodes. Since our focus is on the efficient use of the IB CC, we don't need to display the result of the simulation when the IB CC was disable in its own graph rather we preferred to present and analyse the overall performance of the network in both cases. But the result of the nodes that were sending for the hotspot was as shown in Figure 4.10. The performance of the nodes were better than the one shown in the Figure 4.8 which even

reached about 300MBps at the beginning. This is due to the fact that in this scenario only half (50%) of the nodes were sending traffic which make the network less congested than the previous one in which 100% of them were sending. The victim node *H512* was out performing than any of the others as shown in 'gray' line since it was using a less congested link to send for its destination node *H542* while the rest sent for the same destination.

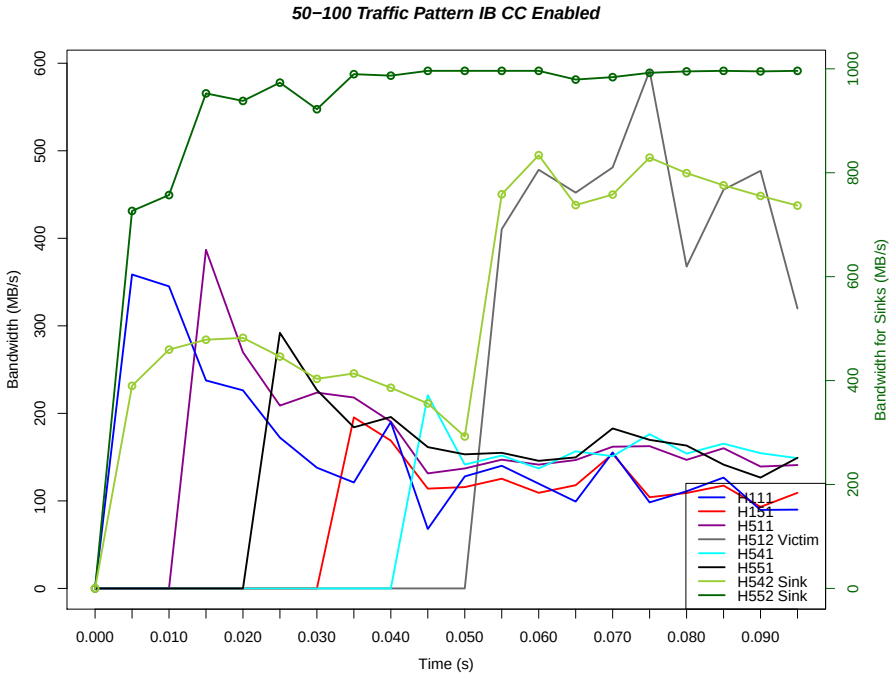

Figure 4.10: 50%-100% Uniform Random Traffic Pattern.

When we see the overall performance of the network for the two cases as shown in Figure 4.11, the average throughput was 18Gbps when the IB CC was enabled and the graph was by far above the disabled one which had average throughput of 13Gbps. Here, it shouldn't be expected that the average throughput of the two cases in this scenario to be the same as the above shown in Figure 4.9 because the number of nodes generating traffic in this scenario was only half of the one used above.

Figure 4.11: 50%-100% Uniform Random Traffic Pattern.

4.3.3 100%-50% Uniform Random Traffic Pattern

This section presents and analysis the simulation results obtained when 100% of the nodes sent for a node chosen randomly from 50% of the nodes in the network, in addition to the contributors and the victim nodes. As expected, the congestion was the worst in this scenario because of the number of nodes sending traffic are twice that of the destination nodes. As shown in Figure 4.12, the far corner nodes such as *H111* and *H151* were largely suffered by the congestion and their graph near to the bottom line. Since all the links was congested in this scenario, even the victim node *H512* which sending to a different destination had no a significantly better performance than the other nodes (*H511, H541, H551*) at the bottom the topology. This is an indication that the IB CC is not able to handle the situation.

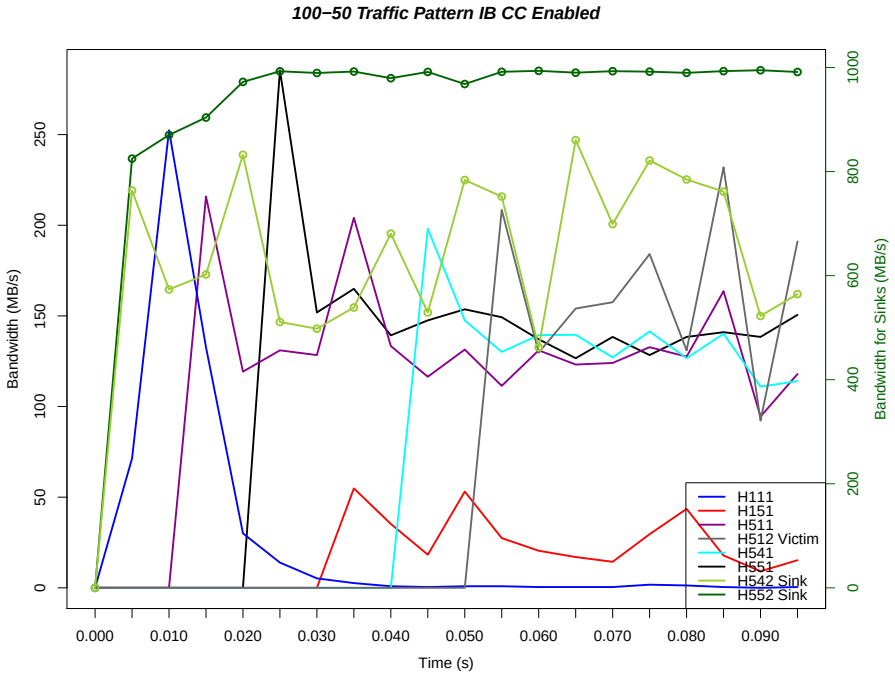

Figure 4.12: 100%-50% Uniform Random Traffic Pattern.

Figure 4.13 shows the overall performance of the network. In this scenario the average throughput was 15.7Gbps when the IB CC was enabled and 13Gbps when it was off. The network performance was declined very much despite the IB CC was enabled compared to the above experiments.

Figure 4.13: 100%-50% Uniform Random Traffic Pattern.

4.3.4 Uniform Random Traffic Pattern with Three Hotspots

This scenario presents the results obtained when 30% of the nodes were sending for three hotspots (10% for each hotspot) and the remaining nodes were sending for a node randomly selected from 100% of the nodes in the network. In this scenario, the average overall throughput of the network was 19.8Gbps when the IB CC was enabled and 15.9Gbps for the other case.

Figure 4.14: Uniform Random Traffic Pattern with Three Hotspots.

4.3.5 Traffic Patterns Result Summary

The discussion in this section focuses on the performance of individual nodes and the overall throughput improvements of the network by using the IB CC, postponing the discussion concerning what parameter values to use for the next section. Therefore, the results of the above set of experiments can be summarized from two angles: One from the performance of individual nodes and the other is from the overall throughput improvements of the network.

The performance of the individual nodes depends on the position of the node in the network. The nodes which were near to the destination nodes in a highly congested network had a better performance than the one that were far from the destination in the above scenarios. This was due to the fact that in DOR routing algorithm the links on the right side would be largely congested since all nodes must send packet first in the x-direction to the destination node in the bottom-right corner. As a result, contributor nodes such as *H111* which was located at the opposite side of the destination node *H552* were highly affected by the congestion. This is also one of the reason for the performance degradation of a mesh network topology due to phenomenal increased in the diameter of the topology.

The overall throughput improvements of the network when the IB CC was enabled are summarized in Table 4.2. These values were calculated from the the results obtained in the above experiments as depicted in the Figures 4.9, 4.11, 4.13 and 4.14

Experiments	Performance Improvements (%)
ExperimentC1	25.0
ExperimentC2	38.5
ExperimentC3	20.8
ExperimentC4	24.5

Table 4.2: Performance Improvements.

As shown in Table 4.2, the throughput of the network in *ExperimentC2* was the best one from all the scenarios. This was because the number of nodes generating traffic in this scenario was smaller (50%) than any other scenarios so that the congestion was relatively less in the network. On the other hand, the throughput in *ExperimentC3* was the least from all. This is due to the less number of receiving nodes (50%) which unable to consume the traffic in the network generated by all (100%) of the nodes in the network. However, this traffic pattern was not chosen for the next section to evaluate the parameter values since there was a big oscillation on the performance. As a result, the next least performing and relatively stable traffic pattern *ExperimentC4* was chosen for the next section.

4.4 Finding Proper IB CC Parameter Values

In the previous experiments, we have learned two major things. The first one was how the IB CC is able to avoid the negative effects of congestion such as HOL blocking and parking lot problem, and the consequent results of the overall network performance improvement. Secondly, we learned that how the performance of the network and the effectiveness of the the IB CC are affected by different traffic patterns. This means that different traffic patterns have different rooms for improvements.

In this section, we present and analyse the simulation results of the experiments described in 3.6.4. The traffic pattern and topology chosen for this experiment was the uniform random traffic pattern with three hotspots designed in Figure 3.10. The main goal of this section was to find the proper parameter values through extensive simulation testing by varying the values of two influential IB CC parameters, *Marking_Rate* and *CCTI_Timer*, taken from the switch and the CA, respectively. The simulation results are presented and analysed in *five* scenarios below.

Table 4.3 consists of the values of the two parameters used. The first column lists the *CCTI_Timer* values and the first row entries are the values of the *Marking_Rate*. A total of 30 simulation results are presented and analysed in this section alone.

CCTI_Timer	Marking_Rate					
	1	10	20	30	40	50
20	✓	✓	✓	✓	✓	✓
250	✓	✓	✓	✓	✓	✓
500	✓	✓	✓	✓	✓	✓
750	✓	✓	✓	✓	✓	✓
1000	✓	✓	✓	✓	✓	✓

Table 4.3: Selected IB CC Parameter Values

4.4.1 Scenario I: CCTI_Timer=20

Figure 4.15 presents the time interval overall performances of the network for six simulation results when *CCTI_Timer=20* and six values of the *Marking_Rate*.

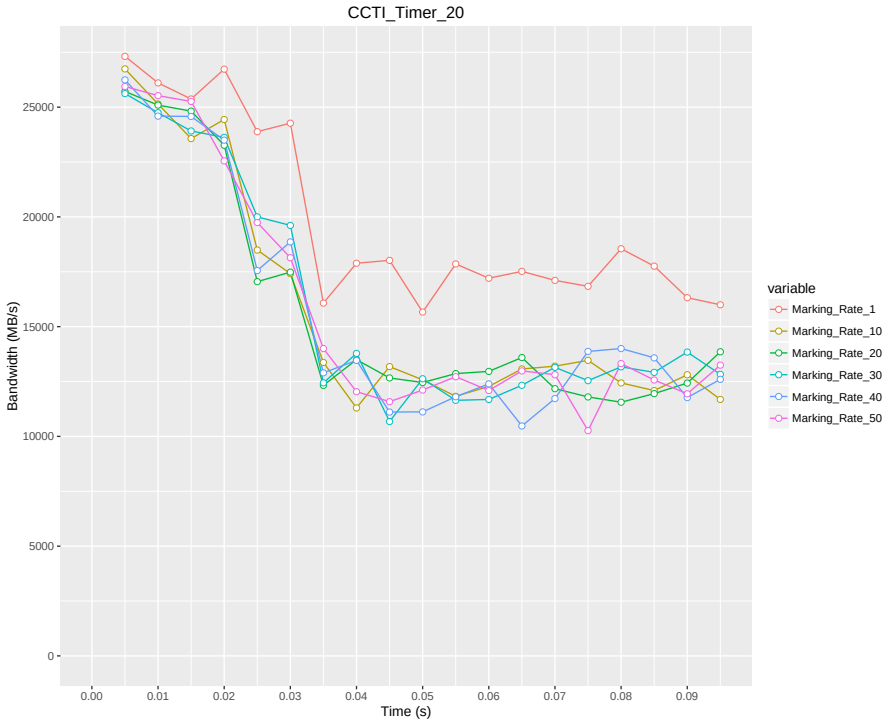

Figure 4.15: CCTI_Timer Value 20 µs

The graphs show that at beginning of the simulation, the network throughput was very high, but it degraded dramatically until the network traffic flow was get stable. This is because, at the beginning, the bandwidth allocation might not reach at equilibrium and the achieved throughput changes over time since the flow control and IB CC mechanism reaction to the network loads may not be effected. After the network was stabilized or reached at equilibrium around *T=0.05sec*, its throughput continued almost constant for all the *six Marking_Rate* values.

As depicted with the graph in 'red' line graph in Figure 4.15, the throughput of the network was out performed better with average throughput around 19.8Gbps when the *Marking_Rate=1* which was similar to the graph shown in Figure 4.14 when the IB CC was enabled. For the rest of the *Marking_Rate* values, it was difficult to distinguish one from the other, they almost behaved the same manner.

In this scenario, as shown in Table 4.4, when the *Marking_Rate=1* the IB CC efficiency was improved but the rest values was like the IB CC was disabled.

Marking_Rate Values	**Average Throughput (Gbps)**
1	19.8
10	15.0
20	14.9
30	15.1
40	14.8
50	14.9

Table 4.4: CCTI_Timer = 20 µs Average Throughput

4.4.2 Scenario II: CCTI_Timer=250

In this scenario, the *CCTI_Timer* value was incremented from 20 to 250 and tested with all the *six* values of the *Marking_Rate*.

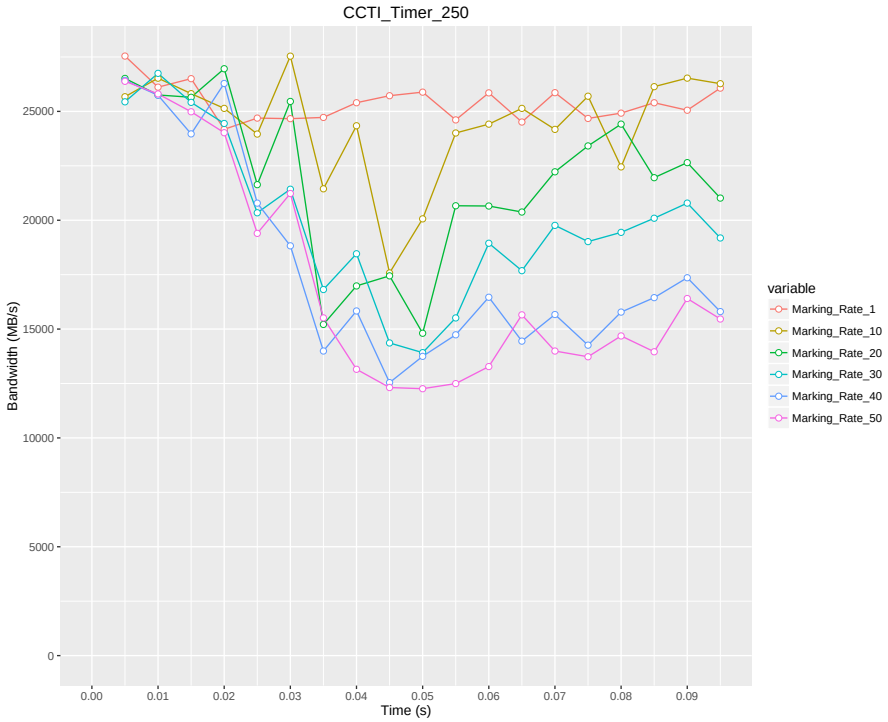

Figure 4.16: CCTI_Timer Value 250 µs

At the very beginning, as shown in Figure 4.16, all the graphs except the 'red' line showed a similar behavior of moving downward until *T=0.05sec* because the network might not be stable at the beginning. After the network was stabilized, the throughput started reviving again in all the cases. However, for the *Marking_Rate=1*, the throughput was almost stable with average throughput of 24.1Gbps and most of the time above the rest of the graphs. The other interesting behavior observed here was the graph of the smaller *Marking_Rate* was most of the time above the larger value *Marking_Rate* graph.

Table 4.5 lists the average throughput for each *Marking_Rate* values. Similar to the scenario above, *Marking_Rate=1* outperformed better than the others and even by far better than the above for the same *Marking_Rate*. In addition, in this scenario, it seems that the throughput and the *Marking_Rate* are inversely proportional. Which means when the value *Marking_Rate* increased, the throughput decreased. This is due to the fact that when the *Marking_Rate* is higher, more packets sent unmarked with FECNs, therefore, the injection rate of the contributors will not be reduced. If the injection rate is not reduced, the congestion will be higher and resulted in throughput degradation.

Marking_Rate Values	Average Throughput (Gbps)
1	24.1
10	23.1
20	20.7
30	18.9
40	17.0
50	16.2

Table 4.5: CCTI_Timer = 250 µs Average Throughput

4.4.3 Scenario III: CCTI_Timer=500

In this scenario, the *CCTI_Timer* was incremented by another 250 µs so that *CCTI_Timer*=500 µs and tested with all the given values of the *Marking_Rate*. As shown in Figure 4.17, although the graphs show a similar behavior with the graphs in Figure 4.16 for each time interval, the throughput of *Marking_Rate=1* degraded while the others' throughput increased. Except for *Marking_Rate=1*, similar to *Scenario II*, the smaller the *Marking_Rate*, the better the throughput was.

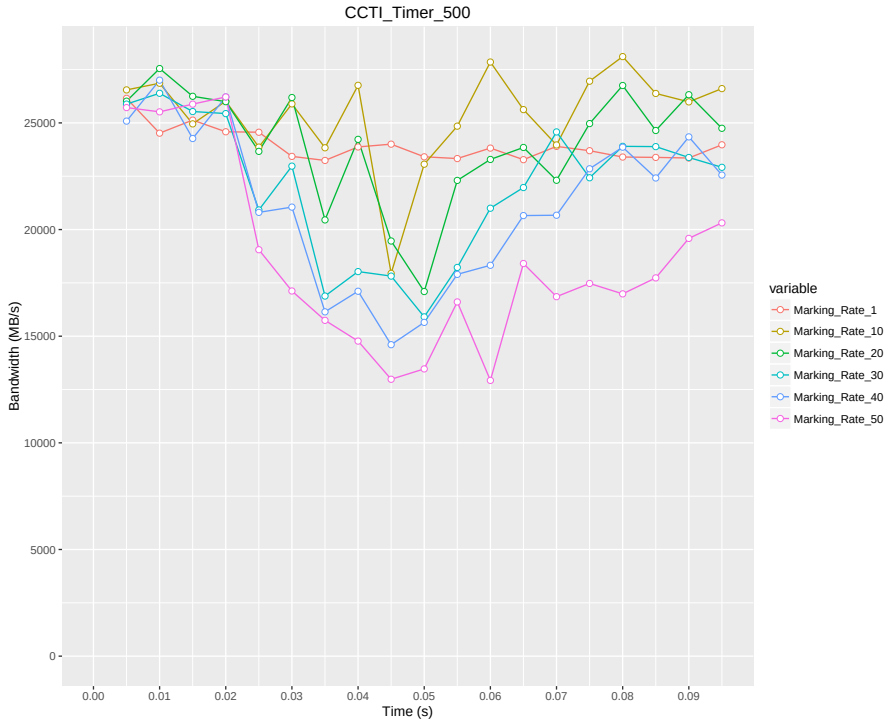

Figure 4.17: CCTI_Timer Value 500 µs

The average throughput for each value of the *Marking_Rate* when *CCTI_Timer=500* are listed in Table 4.6. In this case, the throughput was better when the *Marking_Rate=10*.

Marking_Rate Values	Average Throughput (Gbps)
1	22.8
10	24.1
20	22.8
30	20.9
40	20.1
50	17.7

Table 4.6: CCTI_Timer = 500 µs Average Throughput

4.4.4 Scenario IV: CCTI_Timer=750

In this scenario, the value of the *CCTI_Timer* was incremented from 500 µs to 750 µs and simulated with each values of the *Marking_Rate*. Figure 4.18, the graphs showed a similar behavior as *scenario III* for each time interval, but the graph of *Marking_Rate=1* dropping down while the others are increasing. In this case, the graph with *Marking_Rate=10* was above the rest of the graphs most of the time.

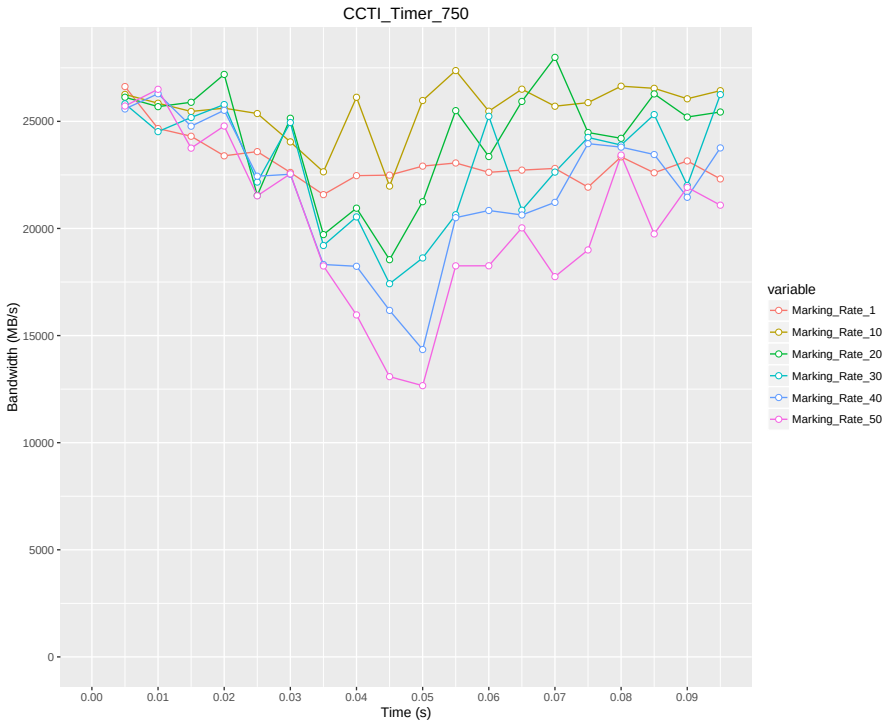

Figure 4.18: CCTI_Timer Value 750 µs

Table 4.7 lists the average throughput for each *Marking_Rate* value. The throughput of the network when the *Marking_Rate=10* was outperformed better similar to the the above scenario.

Marking_Rate Values	Average Throughput (Gbps)
1	22.0
10	24.3
20	23.0
30	21.8
40	20.7
50	19.2

Table 4.7: CCTI_Timer = 750 µs Average Throughput

4.4.5 Scenario V: CCTI_Timer=1000

This scenario presents the simulation results for *CCTI_Timer=1000* with the six values of the *Marking_Rate*. In this setting, the graph for the *Marking_Rate=1* was moved down below all the other values and that of *Marking_Rate=10* came up most of the time than the rest of the graphs.

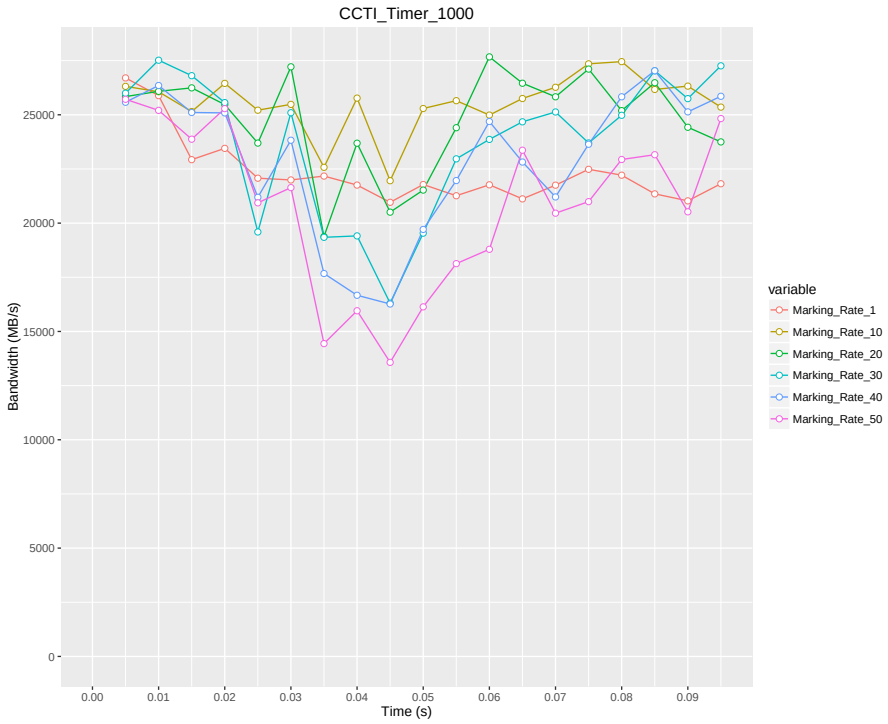

Figure 4.19: CCTI_Timer Value 1000 µs

Table 4.8 lists the average throughput for *CCTI_Timer=1000*. Similar to the above two scenarios, *Marking_Rate=10* outperformed better than the rest.

Marking_Rate Values	Average Throughput (Gbps)
1	21.2
10	24.3
20	23.6
30	22.5
40	21.8
50	19.8

Table 4.8: CCTI_Timer = 1000 µs Average Throughput

4.5 Best Parameter Values Found

Table 4.9 summarises the average throughput of the network resulted from all the combinations of the *Marking_Rate* and *CCTI_Timer* values. Among these results, *four* of them resulted in the highest and nearly similar throughput. These value combinations were when *Marking_Rate=1* and *CCTI_Timer=250*, and *Marking_Rate=10* and for the three values of *CCTI_Timer=500, 750 and 1000*.

CCTI_Timer	Marking_Rate					
	1	10	20	30	40	50
20	19.8	15.0	14.9	15.1	14.8	14.9
250	24.1	23.1	20.7	18.9	17.0	16.2
500	22.8	24.1	22.8	20.9	20.1	17.7
750	22.0	24.3	23.0	21.8	20.7	19.2
1000	21.2	24.3	23.6	22.5	21.8	19.8

Table 4.9: Average Throughput Summary

To realize the performance gains of the IB CC using these four combination of values and to choose the best one from them, it would be nice if we go through the graphs of the network throughput when the *Marking_Rate=1* and *Marking_Rate=10* with all value combinations of the *CCTI_Timer*. Figure 4.20 displayed all the graphs when the *Marking_Rate=1* as well as the graph when the IB CC was disabled which is relevant to visualize by how much the IB CC improved the overall network throughput.

Figure 4.20: Marking_Rate Value 1

As the graphs in 4.20 showed, the throughput of the network was very low when the IB CC was disabled as indicated in a 'purple' line graph. After the IB CC was enabled with *CCTI_Timer=20*, the throughput was improved by 24.5%. The *CCTI_Timer=1000, 750 and 500*, further improved the throughput with 33.3%, 38.4% and 43.4%, respectively. Finally, *CCTI_Timer=250* improved the network from 15.9Gbps when IB CC was disabled to 24.1Gbps, which means the throughput was improved by 51.6%. As a result, the *Marking_Rate=1* and *CCTI_Timer=250* resulted in the highest throughput as shown in the Figure above.

Figure 4.21 depicted the throughput when the *Marking_Rate=10* and when the IB CC was disabled. Even though, the IB CC was enabled with *CCTI_Timer=20*, there was no improvement on throughput as shown in 'red' and 'purple' line graphs. This was an indication that if the IB CC is not configured with the proper parameter values, it will not have effect on the network throughput. When the *CCTI_Timer=250*, the throughput was improved by 45.3%. When *CCTI_Timer=500*, the throughput was improved by 51.6%. Finally, when *CCTI_Timer=750 and 1000*, we obtained almost similar throughput improvements of 52.8%.

Figure 4.21: Marking_Rate Value 10

Despite the fact that the throughput of the network was a bit higher when the *Marking_Rate=10* and the *CCTI_Timer=750 and 1000*, these parameter values are not our choice. This is because as we discussed in Section 2.2, one of the characteristics of an efficient congestion control mechanism is to have *low oscillation* to provide stability in face of high bandwidth or large delay. However, the time interval network throughput for the first cases had a very high oscillation as shown in Figure 4.21. In addition to this, using smaller *Marking_Rate* with low values of the *CCTI_Timer* is usually recommended to achieve the best throughput [15]. As a result, we recommend the parameter values *Marking_Rate=1* and the *CCTI_Timer=250* as a default configuration values for this network topology. These two parameters are also highly correlated to each other. The *Marking_Rate* parameter determines the average number of packets sent unmarked between two consecutive FECNs marked packets at the switch detecting congestion. Every time a BECNs arrives, the HCA lowers its injection rate. The *CCTI_Timer* decides how long the injection rate stays at a certain level before it is increased again. This is the reason why for larger Marking_Rate, the higher values of the *CCTI_Timer* yields a better throughput.

Chapter 5

Discussions

This chapter presents the efforts exerted on this work, the technical challenges during the course of the project, the contributions of the project and finally recommendations for future research directions.

5.1 Overall Project Evaluation

The goal of this project was to explore the InfiniBand congestion control parameters on mesh topologies. The focus is mainly on finding the proper IB CC parameter values that ensure the efficient behavior of the IB CC. This is to improve the performance of a given mesh topology network which utilizes IB CC by setting appropriate parameter values for the IB CC in order to utilize its full capability. While approaching the stated problem statement, we developed python scripts to automate the creation of the network topology model and the forwarding tables based on the DOR routing algorithm. The capability of the IB CC to avoid the negative effects of congestion such as HOL blocking and unfairness was evaluated through a carefully design traffic flow. The performance gains of the IB CC was evaluated by setting different simulation scenarios. The efficiency of the IB CC was also studied using different traffic patterns. The performance influence of the IB CC parameter values was studied using two important parameters (*Marking_Rate* and *CCTI_Timer*) which are chosen from the switch and the channel adapter. The impact of the IB CC, the effect of the traffic patterns and the parameter values was plotted on graphs using supported scripts and analyzed through comparison and reasoning.

The results of the experiments designed in Section 3.6.1 showed that the designed network topology model and forwarding table was implemented properly and the IB CC was also working as intended and able to avoid the negative effect of congestion. The results of the experiments designed in Section 3.6.2 proved that the IB CC was also effective for larger mesh topologies. The results of the experiments designed in Section 3.6.3 indicated that enabling the IB CC introduced better performance gains for different traffic patterns. The majority of the simulation results obtained from the experiments designed in Section 3.6.4 helped us to find the best

parameter values for the chosen mesh topology. These best parameter values found are listed in Table 5.1:

Table 5.1: Best Parameter Values

Best Parameter Values Found
Threshold = 2
Marking_Rate = 1
Packet_Size = 1
CCTI_Timer = 250
CCTI_Limit = 128
CCTI_Min = 0
CCTI_Increase = 1

5.2 Changes from the Initial Plan

At the early stage of the project, we were attempting to explore both mesh and torus topologies and automated the network topology model for both kinds of topologies. However, later we understand that the problem domain is too vast to accomplish in the strict project period. So, we limited ourselves only on 2D and 3D mesh network topologies. For these two kinds of mesh topologies, we automated the creation of the forwarding tables based on the interconnection formulated in the network topology model. The effectiveness of the IB CC was evaluated on 2D mesh topology using different traffic patterns and set of parameter values. The influence of the parameter values on the effectiveness of the IB CC was studied using two important parameters through several simulation tests.

Although the initial plan was changed to limit the scope of the project due to lack of time, the resources such as scripts developed are a general purpose once that can be used to expand the experiments and simulations for further investigations. Since the project is successful accomplished for the chosen parameters, future researches works can be easily established on top of this project.

5.3 Challenges

The time constraint and several technical difficulties are the major challenges while conducting this project. As mentioned above the initial plan was to explore several topologies, traffic patterns and parameters but that couldn't be accomplished due to the strict time boundary of the project. During the course of the project we were challenged by lots of hassles on the configurations of the simulation models.

Large OMNeT++ simulation result vector and index files were the most serious unavoidable challenge of this project. This challenge was started

to happened while we started simulating for different traffic patterns with 100% of the nodes started generating and sending traffic. At this time, the were about 50 nodes and 25 Switches in the network topology model that should either send or receive traffic. When we attempted to run the simulation for a simulation time of *T=0.7sec*, the size of the vector files reached around 80GB and the simulation crashed while only completing 15%. This file size was the size of the disk on VMs with large flavor. Even we tried to attach volumes to VMs, our volume quota was only 120GB which was not even enough to store a single simulation file. If the simulation was allowed to complete to 100% without disk volume restrictions, it might require over 400GB disk for a single simulation. As a result, we had limited the simulation time to *T=0.1sec* for all the simulations after this time onwards. Even with this simulation time, we should extract the relevant values of the vector file and remove the first one before starting the next simulation to free up disk spaces.

Another an expected obstacle was the downtime of the ALTO cloud at a very critical time. As we mentioned in Section 3.2, we used the ALTO cloud to get the benefit of multiple simultaneous simulation runs in several machines at a time. However, after all the VMs created, the OMNeT++ was installed, the IB Model was integrated and the model of the network topology, the simulation configuration, and so on are configured, checked and ready for further simulations, the ALTO cloud was down and reinstalled. Due to this, we were forced to create the environment and reconfigure everything from scratch and have to wait at least for a week to get back the ALTO cloud.

5.4 Contribution of the project

With this work several novel designs, implementations, findings and ideas are proposed. To mention a few:

- The three python scripts developed to generate *ned* file source codes can be used with/wthiout modifications to create any type of 2D mesh, 3D mesh and torus network topology model that can be used in OMNeT++ environment.

- The two python scripts developed based on the above topology model and the DOR routing algorithm to create FDBs can be used with/without modifications to generate forwarding tables for any size of 2D and 3D mesh topology.

- The above two automated tasks will greatly reduce the time-lines of further researches in the area.

- The approaches followed and the experiment designs can be used as a base for further research on mesh topologies.

- The work verified that the effectiveness of the InfiniBand congestion control to avoid the problem of congestion such as unfairness and victim flow due to HOL blocking.
- The best parameter values found can be used as the default parameter values to configure IB CC in many InfiniBand application areas that use mesh topology.

5.5 Future Works

Exploring parameter values need hundreds of exhaustive experiments and simulations. Due to time constraints, we are unable to conduct a more comprehensive study and arrive a more accurate and concrete conclusions. However, the results and findings can be extrapolated for further researches. As a result, a number of recommendations can be proposed for future research directions:

- In this work, only two IB CC parameter values were considered. However, this should be extended to explore the values of other important parameters such as the *Threshold*.
- Using the designed network topology models and forwarding table, further simulations can be made on 3D and torus topologies.
- The study can also be extended to include other topologies, traffic patterns and applications.
- Since the effect of the parameter values depend on the topology and traffic pattern, IB CC parameter values configuration testing and validation is required each time a new topology is designed. However, a mechanism should be formulated to find the optimal parameter values that can be used for a variety of topologies and traffic patterns.

Chapter 6

Conclusion

The main goal of this thesis was to study the effect of the IB CC on the network performance and explore a selected set of IB CC parameter values in order to understand their impact on the behaviour of the IB CC mechanism and evaluate the performance improvements on various scenarios for mesh topology.

The problem statement is addressed via designing and conducting a set of simulations using different traffic patterns and varying the parameter values of selected IB CC parameters. The effect of the IB CC was studied by enabling and disabling the IB CC and studying the performance improvements of the network. The effect of the IB CC parameters was studied by varying the IB CC parameter values and analysing the results to ensure the performance gains of the network.

The analysis of the simulation results revealed that the IB CC mechanism avoided the negative effect of congestion such as unfair share of network resources due to the parking lot problems and victim flows due to HOL blocking. The effect of the IB CC parameters result analysis revealed that the two IB CC parameter values *Marking_Rate=1* and *CCTI_Timer=250* improved the network performance by 51.6% for a given mesh topology.

This work finds very promising results. However, further researches are required to exhaustively explore and study all the parameter spaces and other topologies to further improve the efficiency of the IB CC.

Bibliography

[1] Mehdi Baboli, NS Husin, and MN Marsono. "A comprehensive evaluation of direct and indirect network-on-chip topologies." In: *Proceedings of the 2014 international conference on industrial engineering and operations management*. 2014, pp. 2081–2090.

[2] Rajendra V Boppana and Suresh Chalasani. "Fault-tolerant wormhole routing algorithms for mesh networks." In: *Computers, IEEE Transactions on* 44.7 (1995), pp. 848–864.

[3] Per Nikolaj D Bukh and Raj Jain. *The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling*. 1992.

[4] Thomas Chamberlain. *Learning OMNeT++*. Packt Publishing Ltd, 2013.

[5] William J Dally and Charles L Seitz. "Deadlock-free message routing in multiprocessor interconnection networks." In: *Computers, IEEE Transactions on* 100.5 (1987), pp. 547–553.

[6] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[7] *Download Ubuntu Desktop*. "http : / / www . ubuntu . com / download / desktop". Accessed: 2016-01-15.

[8] Jose Duato, Sudhakar Yalamanchili, and Lionel M Ni. *Interconnection networks: An engineering approach*. Morgan Kaufmann, 2003.

[9] Jose Duato et al. "A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks." In: *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE. 2005, pp. 108–119.

[10] Jesus Escudero-Sahuquillo et al. "Efficient and cost-effective hybrid congestion control for HPC interconnection networks." In: *Parallel and Distributed Systems, IEEE Transactions on* 26.1 (2015), pp. 107–119.

[11] J Flich et al. "VOQ SW: a methodology to reduce HOL blocking in InfiniBand networks." In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE. 2003, 10–pp.

[12] Patrick Geoffray and Torsten Hoefler. "Adaptive routing strategies for modern high performance networks." In: *High Performance Interconnects, 2008. HOTI'08. 16th IEEE Symposium on*. IEEE. 2008, pp. 165–172.

[13] Ernst Gunnar Gran and Sven-Arne Reinemo. "InfiniBand congestion control: modelling and validation." In: *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2011, pp. 390–397.

[14] Ernst Gunnar Gran et al. "Exploring the scope of the InfiniBand congestion control mechanism." In: *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE. 2012, pp. 1131–1143.

[15] Ernst Gunnar Gran et al. "First experiences with congestion control in InfiniBand hardware." In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–12.

[16] Ernst Gunnar Gran et al. "On the relation between congestion control, switch arbitration and fairness." In: *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE. 2011, pp. 342–351.

[17] Mitchell Gusat et al. "Congestion control in InfiniBand networks." In: *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*. IEEE. 2005, pp. 158–159.

[18] Marius Hillenbrand et al. "Virtual InfiniBand clusters for HPC clouds." In: *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*. ACM. 2012, p. 9.

[19] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. "Optimized routing for large-scale infiniband networks." In: *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*. IEEE. 2009, pp. 103–111.

[20] *InfiniBand Trade Association, InfiniBand Architecture Specification, Volume 1, Release 1.2*. "http://www.infinibandta.org/specs/". Accessed: 2015-02-15.

[21] Cruz Izu, José Miguel-Alonso, and José A Gregorio. "Evaluation of interconnection network performance under heavy non-uniform loads." In: *Distributed and Parallel Computing*. Springer, 2005, pp. 396–405.

[22] Dina Katabi, Mark Handley, and Charlie Rohrs. "Congestion control for high bandwidth-delay product networks." In: *ACM SIGCOMM Computer Communication Review* 32.4 (2002), pp. 89–102.

[23] Parviz Kermani and Leonard Kleinrock. "Virtual cut-through: A new computer communication switching technique." In: *Computer Networks (1976)* 3.4 (1979), pp. 267–286.

[24] Mohammad Ayoub Khan and Abdul Quaiyum Ansari. "Quadrant-based XYZ dimension order routing algorithm for 3-D Asymmetric Torus Routing Chip (ATRC)." In: *Emerging Trends in Networks and Computer Communications (ETNCC), 2011 International Conference on*. IEEE. 2011, pp. 121–124.

[25] HT Kung and Robert Morris. “Credit-based flow control for ATM networks.” In: *IEEE network* 9.2 (1995), pp. 40–48.

[26] Charles E Leiserson. “Fat-trees: universal networks for hardware-efficient supercomputing.” In: *Computers, IEEE Transactions on* 100.10 (1985), pp. 892–901.

[27] Xuan-Yi Lin, Yeh-Ching Chung, and Tai-Yi Huang. “A multiple LID routing scheme for fat-tree-based InfiniBand networks.” In: *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE. 2004, p. 11.

[28] *Linux and Unix grep command*. "http://www.computerhope.com/unix/ugrep.htm". Accessed: 2016-03-15.

[29] Ahmed Yusuf Mahamud. *Exploring InfiniBand Congestion Control*. Master Thesis: submitted to University of Oslo, 2015.

[30] Viktor Mauch, Marcel Kunze, and Marius Hillenbrand. “High performance cloud computing.” In: *Future Generation Computer Systems* 29.6 (2013), pp. 1408–1416.

[31] *OMNEST: High-Performance Simulation for All Kinds of Networks, Simulation Models*. "https://omnest.com/". Accessed: 2015-02-27.

[32] *OMNeT++: Discrete Event Simulator*. "https://omnetpp.org/". Accessed: 2015-02-27.

[33] *Oslo and Akershus University College of Applied Sciences: HiOA is getting its own cloud*. "https://www.hioa.no/eng/Aktuelle-saker-fra-2013/HiOA-is-getting-its-own-cloud". Accessed: 2016-01-15.

[34] Fabrizio Petrini and Marco Vanneschi. “k-ary n-trees: High performance networks for massively parallel architectures.” In: *Parallel Processing Symposium, 1997. Proceedings., 11th International*. IEEE. 1997, pp. 87–93.

[35] Gregory F Pfister. “An introduction to the infiniband architecture.” In: *High Performance Mass Storage and Parallel I/O* 42 (2001), pp. 617–632.

[36] G Pfister et al. “Solving hot spot contention using infiniband architecture congestion control.” In: *Proceedings HP-IPC 2005* (2005).

[37] *Project Announcement: R package for OMNeT++*. "https://omnetpp.org/component/content/article?id=3666:project-announcement-r-package-for-omnet". Accessed: 2016-04-30.

[38] Himabindu Pucha, Saumitra M Das, and Y Charlie Hu. “The performance impact of traffic patterns on routing protocols in mobile ad hoc networks.” In: *Computer Networks* 51.12 (2007), pp. 3595–3616.

[39] *Python*. "https://www.python.org/". Accessed: 2016-03-15.

[40] Sanguthevar Rajasekaran, John Reif, and Reda A Ammar. “Hierarchical Performance Modeling and Analysis of Distributed Software Systems.” In: *Handbook of Parallel Computing: Models, Algorithms and Applications*. Chapman and Hall/CRC, 2007, pp. 12–1.

[41] Ville Rantala, Teijo Lehtonen, and Juha Plosila. *Network on chip routing algorithms*. Citeseer, 2006.

[42] *TOP500 LIST, NOVEMBER 2015*. "http://top500.org/lists/2015/11/". Accessed: 2015-02-15.

[43] Paul Torfs and Claudia Brauer. "A (very) short introduction to R." In: *Hydrology and Quantitative Water Management Group, Wageningen University, The Netherlands* (2014).

[44] Yoshio Turner, Jose Renato Santos, and G John Janakiraman. "An approach for congestion control in InfiniBand." In: *HP Laboratories Palo Alto, Internet Systems and Storage Laboratory2001* (2001).

[45] *Tutorial for the omnetpp R package*. "https://github.com/omnetpp/omnetpp-resultfiles/wiki/Tutorial-for-the-omnetpp-r-package". Accessed: 2016-04-30.

[46] Andras Varga. "Omnet++ user manual." In: *OMNeT++ Discrete Event Simulation System. Available at: https://omnetpp.org/doc/omnetpp/manual/* (2010).

[47] András Varga and Rudolf Hornig. "An overview of the OMNeT++ simulation environment." In: *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2008, p. 60.

[48] *Visualizing output scalars and vectors*. "https://omnetpp.org/doc/omnetpp/tictoc-tutorial/part5.html". Accessed: 2016-03-20.

[49] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer Science & Business Media, 2009.

[50] Hadley Wickham. *R packages*. " O'Reilly Media, Inc.", 2015.

[51] Sapna Yadav and C Rama Krishna. "CCTorus: A New Torus Topology for Interconnection Networks." In: *Int'l Conference on Advanced Computational Technologies & Creative Media (ICACTCM'2014) Aug*. 2014, pp. 14–15.

[52] Sapna Yadav, Richa Mishra, and Umesh Gupta. "Performance evaluation of different versions of 2D Torus network." In: *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*. IEEE. 2015, pp. 178–182.

[53] Hans Zima. *High Performance Computing: 4th International Symposium, ISHPC 2002, Kansai Science City, Japan, May 15-17, 2002. Proceedings*. Vol. 4. Springer Science & Business Media, 2002.

Appendices

Appendix A

Scripts

A.1 2DMeshTopo.py

```
1  #!/usr/bin/python
2  from collections import defaultdict
3  class meshTopo:
4   def __init__(self, n=None, m=None):
5                   self.n = n
6    self.m = m
7
8   def importPackages(self):
9    open('2DMeshTopo.ned', 'w').close()
10
11   writePack=open("2DMeshTopo.ned","a")
12   print>>writePack,"package␣networks.meshTopo;\n\
        nimport␣src.HCA;\nimport␣src.IDGenerator;" + \
13                          "\nimport␣src.Switch;\n\
                               nmodule␣meshTopo{\n\t//
                               @display(\"647=298;bgl=3;
                               bgb=643,294,#C0D7E3\");\n\
                               tsubmodules:"
14   writePack.close()
15
16  def addEndNodes(self, node, switch):
17   hca=open('2DMeshTopo.ned', 'a')
18   display3 = 0
19                  for x in range(1, switch+1):
20    display2 = 0
21    for y in range(1, switch+1):
22     display1=0
23                             for z in range(1, node+1):
24      print>>hca, "\t\tH%d%d%d:␣HCA␣{"%(x,y,z) + "\n\t\
           t\t" + "@display(\"p=%d,%d"%(36+display2,100+
           display1+display3)+"\");" + "\n\t\t}"
25      display1 += 60
```

```
26      display2 += 160
27     display3 += 120
28
29    hca.close()
30
31   def addSwitchs(self, switch, node):
32    switchFile=open('2DMeshTopo.ned', 'a')
33
34    # Determine number of Ports and Position of the
         switchs
35    display2 = 0
36    for x in range(1, switch+1):
37     display1 = 0
38     for y in range(1, switch+1):
39      if ((x==y==1) or (x==y==switch) or (x==switch and
          y==1) or (x==1 and y==switch)):
40       port=2+node
41      elif(((x==1 or x==switch) and (y!=1 or y!=switch))
          or((y==1 or y==switch) and (x!=1 or x!=switch))
          ):
42       port=3+node
43      else:
44       port=4+node
45
46
47      print>>switchFile, "\t\tSW%d%d: Switch {"%(x,y)  +
           "\n\t\t\t" + \
48      "parameters:" + "\n\t\t\t\t" + "numSwitchPorts = %
          d;"%port + "\n\t\t\t\t" + "@display(\"p=%d,%d;i
          =abstract/switch"%(120+display1,120+display2)+"
          \");" + "\n\t\t\t" +\
49      "gates:" + "\n\t\t\t\t" + "out[%d];"%port + "\n\t\
          t\t\t" + "in[%d];"%port + "\n\t\t}"
50      display1 += 160
51     display2 += 120
52    switchFile.close()
53
54   def addGenerator(self, node,switch):
55    genFile=open('2DMeshTopo.ned', 'a')
56    print>>genFile, "\t\tIDGen: IDGenerator {" + "\n\t\t
         \t" + "@display(\"p=236,30;i=abstract/multicast
         \");" + "\n\t\t\t" + "gates:" +  "\n\t\t\t\t" + "
         gate[%d];"%(node*(switch**2)) + "\n\t\t}"
57    genFile.close()
58
59   def connections(self, switch, node):
60    connFile=open('2DMeshTopo.ned', 'a')
61
```

```
62      #Create Dict and List to hold switch information
63    switchDict = defaultdict(dict)
64    ports = []
65          for x in range(1, switch+1):
66              for y in range(1, switch+1):
67                  if ((x==y==1) or (x==y==switch) or (x
                       ==switch and y==1) or (x==1 and y==
                       switch)):
68                      port=2+node
69                          if port not in ports:
70                              ports.append(port)
71                              switchDict[x][y]=port
72                  elif(((x==1 or x==switch) and (y!=1 or
                        y!=switch))or((y==1 or y==switch)
                       and (x!=1 or x!=switch))):
73                      port=3+node
74                      if port not in ports:
75                          ports.append(port)
76                          switchDict[x][y]=port
77                  else:
78                          port=4+node
79                      if port not in ports:
80                          ports.append(port)
81                          switchDict[x][y]=port
82
83    # Create connections
84    delay0 = "␣-->␣{delay␣=␣5ns;␣}␣-->␣"
85    delay = "␣-->␣{␣␣delay␣=␣5ns;@display(\"1s=red,2\");
         ␣}␣-->␣"
86    print>>connFile,"\tconnections:"
87    for x, values in switchDict.iteritems():
88     for y, port in values.iteritems():
89
90      #Connect Switches to End Nodes
91      for i in range(1, node+1):
92       #Switch to HCA
93       print>>connFile,"\t\tH%d%d%d.out"%(x,y,i) +
            delay0 + "SW%d%d.in[%d];"%(x,y,i-1)
94       print>>connFile,"\t\tSW%d%d.out[%d]"%(x,y,i-1) +
            delay0 + "H%d%d%d.in;"%(x,y,i)
95
96      #Connect Switches to Switches
97      k = 0
98      c = 1
99      if(x==y==1):
100      for j in range(node, min(ports)):
101       print>>connFile,"\t\tSW%d%d.out[%d]"%(x,y,j+c) +
              delay + "SW%d%d.in[%d];"%(x+k,y+c+k,j-k)
```

```
102         print>>connFile,"\t\tSW%d%d.out[%d]"%(x+k,y+c+k,
                j-k) + delay + "SW%d%d.in[%d];"%(x,y,j+c)
103                                 #swap values
104                                 k = c
105         c = c-2
106       print>>connFile,"\n"
107     elif(x==1 and 1<y<switch):
108       for j in range(node, min(ports)):
109           print>>connFile,"\t\tSW%d%d.out[%d]"%(x,y,j+c
                +k) + delay + "SW%d%d.in[%d];"%(x+k,y+c,j+
                k)
110           print>>connFile,"\t\tSW%d%d.out[%d]"%(x+k,y+c
                ,j+k) + delay + "SW%d%d.in[%d];"%(x,y,j+c+
                k)
111           #swap values
112           k,c = c,k
113       print>>connFile,"\n"
114     elif(x==1 and y==switch):
115       print>>connFile,"\t\tSW%d%d.out[%d]"%(x,y,min(
              ports)-1) + delay + "SW%d%d.in[%d];"%(x+c,y,
              min(ports)-1)
116       print>>connFile,"\t\tSW%d%d.out[%d]"%(x+c,y,min(
              ports)-1) + delay + "SW%d%d.in[%d];"%(x,y,min(
              ports)-1)
117       print>>connFile,"\n"
118     elif(1<x<switch and y==1):
119       for j in range(node, min(ports)):
120         print>>connFile,"\t\tSW%d%d.out[%d]"%(x,y,j+c+k)
              + delay + "SW%d%d.in[%d];"%(x+k,y+c,j-k)
121         print>>connFile,"\t\tSW%d%d.out[%d]"%(x+k,y+c,j-
              k) + delay + "SW%d%d.in[%d];"%(x,y,j+c+k)
122         #swap values
123         k,c = c,k
124       print>>connFile,"\n"
125     elif(x==switch and y<switch):
126       print>>connFile,"\t\tSW%d%d.out[%d]"%(x,y,min(
              ports)-1) + delay + "SW%d%d.in[%d];"%(x,y+c,
              min(ports)-2)
127       print>>connFile,"\t\tSW%d%d.out[%d]"%(x,y+c,min(
              ports)-2) + delay + "SW%d%d.in[%d];"%(x,y,min(
              ports)-1)
128       print>>connFile,"\n"
129     elif(1<x<switch and y==switch):
130           print>>connFile,"\t\tSW%d%d.out[%d]"%(x,y,min(
                  ports)) + delay + "SW%d%d.in[%d];"%(x+c,y,
                  min(ports)-1)
131           print>>connFile,"\t\tSW%d%d.out[%d]"%(x+c,y,
                  min(ports)-1) + delay + "SW%d%d.in[%d];"%(x
```

```
                ,y,min(ports))
132         print>>connFile,"\n"
133        elif(x==y==switch):
134         print ""
135        else:
136         for j in range(node, min(ports)):
137             print>>connFile,"\t\tSW%d%d.out[%d]"%(x,y,j+c
                    +k) + delay + "SW%d%d.in[%d];"%(x+k,y+c-k,
                    j+k)
138             print>>connFile,"\t\tSW%d%d.out[%d]"%(x+k,y+c
                    -k,j+k) + delay + "SW%d%d.in[%d];"%(x,y,j+
                    c+k)
139             #swap values
140             k = c
141         print>>connFile,"\n"
142     #Generator to HCA
143     print>>connFile,"\n"
144     print>>connFile,"\t\t//HCAs need IDs in case of
            parallel simulations"
145             counter=0
146     for x, values in switchDict.iteritems():
147                             for y, port in values.
                                    iteritems():
148       for i in range(1, node+1):
149           print>>connFile,"\t\tIDGen.gate[%d]"%counter +
                  "<--> { delay = 5ns; } <--> " + "H%d%d%d.
                  gateIDGen;"%(x,y,i)
150         counter += 1
151     connFile.close()
152
153   def addFabric(self):
154                     writeFabric=open("2DMeshTopo.ned","a")
155                     print>>writeFabric,"}\nnetwork FABRIC
                        extends meshTopo" + "\n{\n\t" + "
                        parameters:" + "\n\tsubmodules:\n}"
156     writeFabric.close()
157 #main funcltion
158  if __name__ == '__main__':
159   #Enter Number of Nodes and Switchs
160   nNodes= raw_input('Enter Number of End Nodes at a
          Switch:')
161   nSwitch= raw_input('Enter the Number of Switchs in
          one Dimension for 2D Mesh Topology:')
162           try:
163    #Import packages
164    meshObj = meshTopo()
165    meshObj.importPackages()
166
```

```
167             #convert to Integer
168       numNode = int(nNodes)
169       numSwitch = int(nSwitch)
170
171       #Add HCAs
172       meshObj.addEndNodes(numNode,numSwitch)
173
174       #Add Switchs
175       meshObj.addSwitchs(numSwitch,numNode)
176
177       #Add Generator
178       meshObj.addGenerator(numNode,numSwitch)
179
180       #Create links
181       meshObj.connections(numSwitch,numNode)
182
183       #Create Fabric
184       meshObj.addFabric()
185
186     except ValueError:
187         print("Please␣Enter␣a␣Valid␣Number")
```

A.2 2DMeshRouting.py

```
1
2  #!/usr/bin/python
3  class meshRouting:
4      def __init__(self, n=None, m=None):
5          self.n = n
6          self.m = m
7
8      def populateRoutes(self,node,switch):
9          #Create fdbs File
10         open('2DMeshRouting.fdbs', 'w').close()
11
12     routingTable = {}
13     for x in range(1, switch+1):
14           for y in range(1, switch+1):
15         #If the LID doesn't exist
16         routingTable["%d%d"%(x,y)]=[255]
17
18     if(x==y==1):
19       for n in range(1, switch+1):
20        for m in range(1, switch+1):
21         if(x==n and y==m):
22          for z in range(0, node):
23          routingTable.setdefault("%d%d"%(x,y),[]).
                append(z)
```

```
24          elif(m==1):
25           for k in range(0, node):
26               routingTable.setdefault("%d%d"%(x,y) ,[]).
                     append(node)
27          else:
28           for k in range(0, node):
29               routingTable.setdefault("%d%d"%(x,y) ,[]).
                     append(node+1)
30
31      elif(x==1 and 1<y<switch):
32       for n in range(1, switch+1):
33           for m in range(1, switch+1):
34               if(x==n and y==m):
35                   for z in range(0, node):
36                       routingTable.setdefault("%d%d"%(x
                             ,y) ,[]).append(z)
37               elif(m<y):
38                   for k in range(0, node):
39                       routingTable.setdefault("%d%d"%(x
                             ,y) ,[]).append(node)
40               elif(m>y):
41                for k in range(0, node):
42                   routingTable.setdefault("%d%d"%(x,y)
                         ,[]).append(node+1)
43               else:
44                for k in range(0, node):
45                   routingTable.setdefault("%d%d"%(x,y)
                         ,[]).append(node+2)
46      elif(x==1 and y==switch):
47       for n in range(1, switch+1):
48           for m in range(1, switch+1):
49               if(x==n and y==m):
50                   for z in range(0, node):
51                       routingTable.setdefault("%d%d"%(x
                             ,y) ,[]).append(z)
52               elif(m<y):
53                   for k in range(0, node):
54                       routingTable.setdefault("%d%d"%(x
                             ,y) ,[]).append(node)
55               else:
56                for k in range(0, node):
57                   routingTable.setdefault("%d%d"%(x,y)
                         ,[]).append(node+1)
58
59      elif(1<x<switch and y==1):
60       for n in range(1, switch+1):
61           for m in range(1, switch+1):
62               if(x==n and y==m):
```

```
63                  for z in range(0, node):
64                      routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(z)
65              elif(n<x and m==1):
66                  for k in range(0, node):
67                  routingTable.setdefault("%d%d"%(x,y)
                        ,[]).append(node)
68              elif(n>x and m==1):
69               for k in range(0, node):
70                  routingTable.setdefault("%d%d"%(x,y)
                        ,[]).append(node+2)
71              else:
72               for k in range(0, node):
73                  routingTable.setdefault("%d%d"%(x,y)
                        ,[]).append(node+1)
74
75      elif(x==switch and y==1):
76       for n in range(1, switch+1):
77           for m in range(1, switch+1):
78               if(x==n and y==m):
79                   for z in range(0, node):
80                       routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(z)
81               elif(n<x and m==1):
82                   for k in range(0, node):
83                       routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(node)
84               else:
85                   for k in range(0, node):
86                       routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(node+1)
87
88       elif(x==switch and 1<y<switch):
89       for n in range(1, switch+1):
90           for m in range(1, switch+1):
91               if(x==n and y==m):
92                   for z in range(0, node):
93                       routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(z)
94               elif(m<y):
95                   for k in range(0, node):
96                       routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(node)
97               elif(m>y):
98                   for k in range(0, node):
99                       routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(node+1)
100              else:
```

```
101                 for k in range(0, node):
102                     routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(node+2)
103
104     elif(1<x<switch and y==switch):
105      for n in range(1, switch+1):
106          for m in range(1, switch+1):
107              if(x==n and y==m):
108                  for z in range(0, node):
109                      routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(z)
110              elif(n<x and m==switch):
111                  for k in range(0, node):
112                      routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(node+1)
113              elif(n>x and m==switch):
114                  for k in range(0, node):
115                      routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(node+2)
116             else:
117              for k in range(0, node):
118                  routingTable.setdefault("%d%d"%(x,y)
                        ,[]).append(node)
119
120     elif(x==y==switch):
121      for n in range(1, switch+1):
122          for m in range(1, switch+1):
123              if(x==n and y==m):
124                  for z in range(0, node):
125                      routingTable.setdefault("%d%d"%(x
                            ,y),[]).append(z)
126              elif(m==switch):
127                  for k in range(0, node):
128                  routingTable.setdefault("%d%d"%(x,y)
                        ,[]).append(node+1)
129             else:
130              for k in range(0, node):
131                  routingTable.setdefault("%d%d"%(x,y)
                        ,[]).append(node)
132     else:
133         for n in range(1, switch+1):
134             for m in range(1, switch+1):
135                 if(x==n and y==m):
136                     for z in range(0, node):
137                         routingTable.setdefault("%d%d"
                                %(x,y),[]).append(z)
138                 elif(m<y):
139                     for k in range(0, node):
```

```
140                             routingTable.setdefault("%d%d"
                                    %(x,y),[]).append(node)
141                     elif(m>y):
142                         for k in range(0, node):
143                             routingTable.setdefault("%d%d"
                                    %(x,y),[]).append(node+1)
144                     elif(m==y and n<x):
145                         for k in range(0, node):
146                             routingTable.setdefault("%d%d"
                                    %(x,y),[]).append(node+2)
147             else:
148                 for k in range(0, node):
149                     routingTable.setdefault("%d%d"%(x,y)
                            ,[]).append(node+3)
150     #Open the fdbs file
151     writeRoutes=open("2DMeshRouting.fdbs","a")
152
153     #Write sorted route to a file
154     for key in sorted(routingTable.iterkeys()):
155      print("%s:␣%s"%(key,"␣".join(str(x) for x in
            routingTable[key])))
156      print>>writeRoutes,("%s:␣%s"%(key,"␣".join(str(x)
            for x in routingTable[key])))
157
158     writeRoutes.close()
159
160 #main funcltion
161 if __name__ == '__main__':
162      #Enter Number of Nodes and Switchs
163      nNodes= raw_input('Enter␣the␣Number␣of␣End␣Nodes␣
             at␣a␣Switch:')
164      nSwitch= raw_input('Enter␣the␣Number␣of␣Switchs␣in
             ␣one␣Dimension␣for␣2D␣Mesh␣Topology:')
165      try:
166          #convert to Integer
167          numNode = int(nNodes)
168          numSwitch = int(nSwitch)
169
170          #create an object of the class
171     meshObj = meshRouting()
172
173     #Populate Routing
174     meshObj.populateRoutes(numNode,numSwitch)
175
176       except ValueError:
177           print("Please␣Enter␣a␣Valid␣Number")
```

Appendix B

Simulation Configuration

B.1 TwoDMeshTopo.ini

```
1  [General]
2
3  ####################################################
4  # Simulation level settings
5  ####################################################
6  sim-time-limit = ${simtime=0.1}s # max number of
       simulation seconds to run
7  network = FABRIC #This line is for Cmdenv
8  print-undisposed = false
9  debug-on-errors = true
10 cmdenv-express-mode = true
11
12 ####################################################
13 # Seeding information
14 ####################################################
15 num-rngs = 3 # Static seed
16 seed-0-mt = 12
17 seed-1-mt = 9
18 seed-2-mt = 7
19
20 ####################################################
21 # InfiniBand link speed and other parameters
22 ####################################################
23 **.creditSize = 64 # number of bytes in one credit
24 **.width = ${width=4}  # link width 4x, 8x, or 12x
25 **.speed = ${speed=2.5} # link speed 2.5, 5.0, or 10.0
26
27 # interface width, should we set to match the link
       width
28 #**vlarb.width = 4
29 # switch core frequency should be 200, 400, and 800
       for SDR, DDR, and QDR respectively
```

```
30 **SW** vlarb.coreFreq_MH = 800
31 # HCAs frequency is the wire frequency, should be 125,
       250, and 500 for SDR, DDR, and QDR respectively
32 **H** vlarb.coreFreq_MH = 250
33
34 ################################################
35 # GENERATOR
36 ################################################
37 **.gen.floodVLs = "0"
38 **.gen.dstHotSpotPerc = 0
39
40 **.H111.gen.genStartTime = 0s
41 **.H511.gen.genStartTime = 1s
42 **.H551.gen.genStartTime = 2s
43 **.H151.gen.genStartTime = 3s
44 **.H541.gen.genStartTime = 4s
45 **.H512.gen.genStartTime = 5s
46
47 #Never start these, must be higher to avoid traffic
       gen
48 **.H112.gen.genStartTime = 10s
49 **.H121.gen.genStartTime = 10s
50 **.H122.gen.genStartTime = 10s
51 **.H131.gen.genStartTime = 10s
52 **.H132.gen.genStartTime = 10s
53 **.H141.gen.genStartTime = 10s
54 **.H142.gen.genStartTime = 10s
55 **.H152.gen.genStartTime = 10s
56
57 **.H211.gen.genStartTime = 10s
58 **.H212.gen.genStartTime = 10s
59 **.H221.gen.genStartTime = 10s
60 **.H222.gen.genStartTime = 10s
61 **.H231.gen.genStartTime = 10s
62 **.H232.gen.genStartTime = 10s
63 **.H241.gen.genStartTime = 10s
64 **.H242.gen.genStartTime = 10s
65 **.H251.gen.genStartTime = 10s
66 **.H252.gen.genStartTime = 10s
67
68 **.H311.gen.genStartTime = 10s
69 **.H312.gen.genStartTime = 10s
70 **.H321.gen.genStartTime = 10s
71 **.H322.gen.genStartTime = 10s
72 **.H331.gen.genStartTime = 10s
73 **.H332.gen.genStartTime = 10s
74 **.H341.gen.genStartTime = 10s
75 **.H342.gen.genStartTime = 10s
```

```
76  **.H351.gen.genStartTime = 10s
77  **.H352.gen.genStartTime = 10s
78
79  **.H411.gen.genStartTime = 10s
80  **.H412.gen.genStartTime = 10s
81  **.H421.gen.genStartTime = 10s
82  **.H422.gen.genStartTime = 10s
83  **.H431.gen.genStartTime = 10s
84  **.H432.gen.genStartTime = 10s
85  **.H441.gen.genStartTime = 10s
86  **.H442.gen.genStartTime = 10s
87  **.H451.gen.genStartTime = 10s
88  **.H452.gen.genStartTime = 10s
89
90  **.H521.gen.genStartTime = 10s
91  **.H522.gen.genStartTime = 10s
92  **.H531.gen.genStartTime = 10s
93  **.H532.gen.genStartTime = 10s
94  **.H542.gen.genStartTime = 10s
95  **.H552.gen.genStartTime = 10s
96
97  ##################
98  # DESTINATION
99  ##################
100 # possible values are: dstLid, dstSeqOnce, dstSeqLoop,
        dstRandom, dstHotSpot, dstBurstHotSpot
101 #**.dstSeqMode = "dstLid"
102
103 # Src Lids
104 include srcID.ini # source LIDs for the HCAs
105
106 # Dst Lids
107 #include dstID.ini # destination LIDs for the HCAs
108
109 # possible values are: dstLid, dstSeqOnce, dstSeqLoop,
        dstRandom, dstHotSpot, dstBurstHotSpot
110 **H**.gen.dstSeqVecFile = "TwoDMeshTopo.dstSeq"
111 include  TwoDMeshTopo.dstSeq.ini
112 **.gen**.dstSeqMode = "dstRandom"
113 **H**.gen.sizeSeqVecFile = ""
114 #**H_**.msgLengthInMTUs=1
115 #**H_**.gen.msgLength=2048
116 **.gen.dstLid = 1
117
118
119 ########################################
120 # TRAFFIC
121 ########################################
```

```
122 **.gen.trafficDist = "trfFlood"
123 #**.gen.dstSeqVecFile = ""
124 #**.gen.sizeSeqVecFile = ""
125 #**.gen.dstSeqIndex = 0
126
127 ########################################
128 # SWITCH
129 ########################################
130 # generated file holding port groups and FDBs vector
      indexes
131 **.SW**.fdbsVecFile = "TwoDMeshTopo.fdbs"
132 include TwoDMeshTopo.fdbs.ini
133
134 ########################################
135 # TESTING (Tuning)
136 ########################################
137 #**gen.srcLid = 1
138 **gen.burstLength = intuniform(20,100)
139 **gen.burstNumber = 128
140 **gen**.interBurstDelay = 0
141
142 ########################################
143 # IBUF
144 ########################################
145 **H**.ibuf.maxStatic* = ${hibuf=1000}
146 **H**.ibuf.totalBufferSize = ${hibuf}
147 #**ibuf.maxStatic0 = 96 # in credits
148 **ibuf.maxStatic* = ${sibuf=1000} #
149 **ibuf.totalBufferSize = ${sibuf} # in credits
150 # speedup
151 **ibuf.maxBeingSent = 1
152
153 **.Victim_Mask = 1
154
155 ################################################
156 # Results Settings
157 ################################################
158 **.scalar-recording = true
159 **.recordVectors = 1 # set to 1 to dump statistics to
      vector file
160 **.logInterval = ${logint=5}ms
161 include ../../src/modules.ini
162
163
164 [Config flood]
165 **gen**.trafficDist = "trfFlood"
166 description = "Simple Run"
167 #**H**.gen.dstSeqVecFile = ""
```

```
168
169 #Fast interswitch link:
170 **SW**.port[2].obuf.width = 4
171 **SW**.port[2].obuf.speed = 10.0
172 **SW**.port[2].ibuf.maxBeingSent = 2
173 **SW**.port[2].ccmgr.Victim_Mask = 0
174
175 **SW**.port[3].obuf.width = 4
176 **SW**.port[3].obuf.speed = 10.0
177 **SW**.port[3].ibuf.maxBeingSent = 2
178 **SW**.port[3].ccmgr.Victim_Mask = 0
179
180 **SW**.port[4].obuf.width = 4
181 **SW**.port[4].obuf.speed = 10.0
182 **SW**.port[4].ibuf.maxBeingSent = 2
183 **SW**.port[4].ccmgr.Victim_Mask = 0
184
185 **SW**.port[5].obuf.width = 4
186 **SW**.port[5].obuf.speed = 10.0
187 **SW**.port[5].ibuf.maxBeingSent = 2
188 **SW**.port[5].ccmgr.Victim_Mask = 0
189
190 **SW**vlarb.coreFreq_MH = 800
191
192 ##################################
193 # Start of hosts:
194 ##################################
195 #include genDelay.ini
196 #To hotspot #1
197 **.H551.gen.genStartTime = 0.0s
198 **.H541.gen.genStartTime = 0.01s
199 **.H211.gen.genStartTime = 0.02s
200 **.H141.gen.genStartTime = 0.03s
201 **.H251.gen.genStartTime = 0.04s
202 **.H212.gen.genStartTime = 0.05s
203
204 #To hotspot #2
205 **.H111.gen.genStartTime = 0.0s
206 **.H121.gen.genStartTime = 0.01s
207 **.H511.gen.genStartTime = 0.02s
208 **.H451.gen.genStartTime = 0.03s
209 **.H441.gen.genStartTime = 0.04s
210 **.H131.gen.genStartTime = 0.05s
211
212 #To hotspot #3
213 **.H151.gen.genStartTime = 0.0s
214 **.H142.gen.genStartTime = 0.01s
215 **.H411.gen.genStartTime = 0.02s
```

```
216 **.H532.gen.genStartTime = 0.03s
217 **.H422.gen.genStartTime = 0.04s
218 **.H132.gen.genStartTime = 0.05s
219
220 #Never start these, must be higher to avoid traffic
        gen
221 **.H112.gen.genStartTime = 0.0s
222 **.H122.gen.genStartTime = 0.0s
223 **.H152.gen.genStartTime = 0.0s
224
225 **.H221.gen.genStartTime = 0.0s
226 **.H222.gen.genStartTime = 0.0s
227 **.H231.gen.genStartTime = 0.0s
228 **.H232.gen.genStartTime = 0.0s
229 **.H241.gen.genStartTime = 0.0s
230 **.H242.gen.genStartTime = 0.0s
231 **.H252.gen.genStartTime = 0.0s
232
233 **.H311.gen.genStartTime = 0.0s
234 **.H312.gen.genStartTime = 0.0s
235 **.H321.gen.genStartTime = 0.0s
236 **.H322.gen.genStartTime = 0.0s
237 **.H331.gen.genStartTime = 0.0s
238 **.H332.gen.genStartTime = 0.0s
239 **.H341.gen.genStartTime = 0.0s
240 **.H342.gen.genStartTime = 0.0s
241 **.H351.gen.genStartTime = 0.0s
242 **.H352.gen.genStartTime = 0.0s
243
244 **.H412.gen.genStartTime = 0.0s
245 **.H421.gen.genStartTime = 0.0s
246 **.H431.gen.genStartTime = 0.0s
247 **.H432.gen.genStartTime = 0.0s
248 **.H442.gen.genStartTime = 0.0s
249 **.H452.gen.genStartTime = 0.0s
250
251 **.H521.gen.genStartTime = 0.0s
252 **.H522.gen.genStartTime = 0.0s
253 **.H531.gen.genStartTime = 0.0s
254 **.H542.gen.genStartTime = 0.0s
255 **.H552.gen.genStartTime = 0.0s
256
257
258 ################################
259 # Generator speed
260 ################################
261 **gen**.interBurstDelay = 0 # Update this one to slow
        down a sender...
```

```
262 **gen**.intraBurstDelay = ${intrabd=26}
263 **gen**.dstHotSpotPerc = 0
264
265 ################################
266 # SINK must not be too  efficient
267 ################################
268 **sink.pciExpWidth = 8
269 **sink.pciExpTransferRate = 5 # 2.5 equals PCI 1.1,
       5.0 equals PCI 2.0, and 8.0 PCI 3.0
270 **sink.hiccupDuration_us = ${hdur=0.0125}
271 **sink.hiccupDelay_us = 0.1
272
273 ############################
274 # Msg and packet sizes:
275 ############################
276 **gen**.msgLength = 65536 # message length in number
       of bytes
277 **gen**.msgLengthInMTUs = 32 #message length in number
        of mtus
278 **.mtu = 34 # number of credits in one mtu
279 **gen**.packetlengthbytes = 2176 # packetlength in
       bytes
280 **gen**.packetlength = 34 # packetlength in number of
       credits
281 **gen**.floodVLs = "0"
282 **gen**.PktLenDist = "2176"
283 **gen**.PktLenProb = "100"
284
285 ##############################################
286 #dynamic hotspot support
287 ##############################################
288 #**H**.gen.dstSeqVecFile = ""
289
290 ##############################################
291 # Congestion Control
292 ##############################################
293 **.cc_enabled = true
294 **.cc_extended_ibuf_log = false
295 **.qp_level = ${qp=true}
296 **.cc_single_thr_per_outport = ${singleThr=true}
297 **.cc_single_thr_devide = ${singleThrDev=false}
298
299 **.Threshold = ${thr=2}
300 **.Reset_Threshold = ${thrlow=3} #Reset_Threshold
       needs to be HIGHER than Threshold
301 **.Marking_Rate = ${mr=40}
302 **.Packet_Size = 1
303 **.CCTI_Timer = ${cctitimer=20}us
```

```
304 **.CCTI_Limit = 128
305 **.CCTI_Min = 0
306 **.CCTI_Increase = ${cctiinc=1}
307 **.CCT_Algorithm = 0 # 1
308 **.CCT_Div = ${cctdiv=95}
309 **.num_nodes = ${nn=50} #..70 step 10}
310
311 # Exporting data to scalar and vector files
312
313 output-vector-file = tempresults5/${resultdir}/${
        confignname}-simt${simtime}-CCDisabled-markrate40.
        vec
314 output-scalar-file = tempresults5/${resultdir}/${
        confignname}-simt${simtime}-CCDisabled-markrate40.
        sca
315
316 #output-vector-file = tempresults/${resultdir}/${
        confignname}-ib${sibuf}_${hibuf}-Thr${thr}-nn${nn}-$
        {cctitimer}us-intrabd${intrabd}-hdur${hdur}-cctdiv$
        {cctdiv}-${logint}ms-simt${simtime}-Vtest2-
        ccti_inc${cctiinc}-mesh.vec
317 #output-scalar-file = tempresults/${resultdir}/${
        confignname}-ib${sibuf}_${hibuf}-Thr${thr}-nn${nn}-$
        {cctitimer}us-intrabd${intrabd}-hdur${hdur}-cctdiv$
        {cctdiv}-${logint}ms-simt${simtime}-Vtest2-
        ccti_inc${cctiinc}-mesh.sca
```