Flexible topologies in multihop networks.

Someshwar Roy Choudhury

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/2121

This online database contains the full-text of PhD dissertations and Masters’ theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Flexible Topologies in Multihop Networks

By

Someshwar Roy Choudhury

A Thesis
Submitted to the College of Graduate Studies and Research through the Faculty of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science at the University of Windsor

Windsor, Ontario, Canada

2000
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-62277-0
Abstract:

In today's environment, as the need for more bandwidth-intensive networking applications such as data browsing, video conferencing, etc increase, so also does the need for high bandwidth-transport network facilities. Optical WDM networks show great promise in handling such high data volume problems, and it is expected that they will form the backbone of the next generation of high volume networks. Multihop networks show the most promise in that they offer the greatest flexibility of design. In this thesis we have attempted to evaluate and modify some existing optical topologies. These topologies are designed to facilitate the modification of existing network structures without significantly disturbing the overall network. They are compared to an existing topology, GEMNET, and their performance measured according to several criteria.
To my mother and father...
Acknowledgments

I take this opportunity to thank my advisor, Dr. Arunita Jaekel for her continuous encouragement and intensive support throughout my graduate studies. I am also grateful to Dr. Subir Bandyopadhyay who, along with Dr. Jaekel, guided me to the completion of this thesis. This work would not have been achieved without their support and advice. I am also grateful to Dr. Peter Tsin and Prof. Philip Alexander for their appreciated suggestions. Finally I want to express my gratitude to the other members of the optical research group for their continuous encouragement and support.
Contents

Abstract .......................................................................................................................... iv
Acknowledgment ......................................................................................................... vi
List of Figures ............................................................................................................... ix
List of Tables ............................................................................................................... x

Chapter 1  Introduction ............................................................................................... 1
  1.1 Motivation ............................................................................................................ 2
  1.2 Problem Outline ................................................................................................ 2
  1.3 Thesis Organization ............................................................................................ 3

Chapter 2  Review of Literature ................................................................................ 5
  2.1 Basics of Lightwave Technology ....................................................................... 5
  2.2 Physical Configurations ...................................................................................... 6
  2.3 Optical Components ............................................................................................ 6
  2.4 Classifications of Optical Networks .................................................................. 7
    2.4.1 Single Hop Networks ..................................................................................... 8
    2.4.2 Multihop Networks ....................................................................................... 9
    2.4.3 Wavelength Routed Networks ..................................................................... 11
    2.4.4 Static vs. Dynamic Routing Schemes .......................................................... 12
  2.5 Optical Networks Based on Regular Topologies ................................................ 12
    2.5.1 Kautz Graph .................................................................................................. 13
    2.5.2 ShuffleNet ..................................................................................................... 14
    2.5.3 Hypercube ..................................................................................................... 16
    2.5.4 de Bruijn Graph ............................................................................................. 18
    2.5.5 Advantages and Disadvantages of Regular Topologies ............................... 21
  2.6 Flexible Topologies .............................................................................................. 22
    2.6.1 GEMNET ....................................................................................................... 22
    2.6.2 Advantages and Disadvantages of GEMNET .............................................. 26
  2.7 Conclusion ........................................................................................................... 27

Chapter 3  The Proposed Flexible Topologies ............................................................. 28
  3.1 Introduction .......................................................................................................... 28
  3.2 Topology A ........................................................................................................... 28
    Interconnection Pattern .......................................................................................... 29
    Addition of nodes ................................................................................................... 29
    Routing Strategy .................................................................................................... 32
  3.3 Topology B ........................................................................................................... 33
    Interconnection Pattern .......................................................................................... 33
    Addition of Nodes ................................................................................................... 34
    Routing Strategy .................................................................................................... 37
  3.4 Topology C ........................................................................................................... 38
    Interconnection Pattern .......................................................................................... 38
    Addition of Nodes ................................................................................................... 39
    Routing Strategy .................................................................................................... 41
List of Figures

Figure 1: A star-based single hop system ............................................. 8
Figure 2: Physical and logical topologies of a 4-node multihop network .... 10
Figure 3: A K(2, 2) Kautz Digraph ..................................................... 14
Figure 4: A (2, 2) ShuffleNet ............................................................ 15
Figure 5: A Binary Hypercube .......................................................... 17
Figure 6: A (2, 3) de Bruijn Graph .................................................... 19
Figure 7: A (2, 5, 2) GEMNET ......................................................... 23
Figure 8: Interconnection topology with d = 2 for (a) N = 5 and (b) N = 6 31
Figure 9: Interconnection topology with d = 2 for (a) N = 6 and (b) N = 7 36
Figure 10: Interconnection topology with d = 2 for (a) N = 6 and (b) N = 7 40
Figure 11: Average Hop Distance .................................................... 54
Figure 12: Packet loss average for N = 27, 31, 45, 48, 59, 64 ................. 55
Figure 13: Packet loss Std. Devn. for N = 27, 31, 45, 48, 59, 64 ............... 56
Figure 14: Comm. Loss Average for N = 27, 31, 45, 48, 59, 64 ................. 57
Figure 15: Comm. Loss std. devn. for N = 27, 31, 45, 48, 59, 64 ............... 58
Figure 16: Queuing delay average for N = 27, 31, 45, 48, 59, 64 ............... 59
Figure 17: Queuing delay std. devn. for N = 27, 31, 45, 48, 59, 64 ............... 59
Figure 18: Average Hop Distance for PR1 & PR3 ................................ 61
Figure 19: Packet Loss Average for N = 27, 55, 81 ................................ 62
Figure 20: Packet Loss Standard Deviation for N = 27, 55, 81 ..................... 63
Figure 21: Comm. Loss Average for N = 27, 55, 81 ................................ 64
Figure 22: Comm. Loss Std. Devn. for N = 27, 55, 81 ............................. 64
Figure 23: Queuing Delay average for N = 27, 55, 81 ............................. 65
Figure 24: Queuing Delay std. devn. for N = 27, 55, 81 ......................... 66
List of Tables

Table 1: Average Hop Distance for PR2 ................................................................. 54
Table 2: Link load characteristics for \( N = 27, 31, 45, 48, 59, 64 \) and \( (P) = 0, 1, 2, 3 \) ... 60
Table 3: Average Hop Distance for PR1 & PR3 ...................................................... 61
Table 4: Link Load characteristics for \( N = 27, 55, 81 \) and edge \( (P) = 0, 1, 2, 3 \) ... 67
Chapter 1  Introduction

Optical fibers have emerged as the most attractive communications medium in today’s high speed data transfer environment. Their properties such as high bandwidth, low attenuation and reliability make it very suitable for constructing real time dynamic networks capable of handling large volumes of data. Several optical network topologies have been proposed which take advantage of the above-mentioned properties. These topologies usually exhibit a symmetrical structure and low diameter, meaning that maximum hop distance between any two nodes is low.

Optical networks can allow several channels to exist concurrently on a single transmission medium. This is known as wavelength-division multiplexing (WDM). These wavelengths can be switched at intermediate nodes, allowing data to take full advantage of the bandwidth of the channel. However, the switching devices used at the nodes are mostly electronic in nature. This gives rise to a speed mismatch at the opto-electronic interfaces, thus hindering the performance of the network. Extensive research has gone into designing networks that are all-optical in nature, in which the switching components also operate in the optical domain. Limited range wavelength converters have appeared, and some all-optical switch architectures have also been proposed.
1.1 Motivation

Based on the way a packet is transmitted through a network, optical networks can be classified into single hop and multihop networks. In single hop networks, packets travel from the source to the destination in one hop while in multihop networks, the packets hop through zero or more intermediate nodes. This thesis is concerned mainly with multihop network structures as they provide scope for transceiver retuning and scalability. Many multihop topologies are based on regular structures, in which the number of nodes is a subset of a fixed set of integers. This proves to be a hindrance in the design of dynamic networks, as the size of the network cannot be scaled arbitrarily. Thus, an obvious problem for optical network researchers would be to design a multihop network that could be scaled arbitrarily, while retaining its properties such as low diameter and regular symmetry.

1.2 Problem Outline

In our work, we study the performance of three scalable topologies proposed by Dr. Subir Bandyopadhyay and Dr. Arunita Jaekel at the University of Windsor and Dr. Abhijeet Sengupta at the University of South Carolina. The basic structure of these topologies are similar to the de Bruijn graph, but they employ a different routing scheme, thus enabling them to be scaled for an arbitrary number of nodes. As mentioned earlier, most of the network topologies investigated so far do not allow dynamic scaling, or allow scaling in
fixed, predetermined increments. An attempt has been made in this thesis to ultimately design some architectures which will allow a network to be scaled arbitrarily with negligible loss of performance.

We use network simulation in our studies and have written some programs to model the network components and the traffic. As a benchmark topology, we have taken an equivalent single-column GEMNET, whose performance is also simulated using our programs. A single column GEMNET can also be scaled for an arbitrary number of nodes. We intend to show that we can add nodes to the network with minimum perturbation of the network, and as the performances are comparable to GEMNET, their degradation while scaling the network is acceptable.

1.3 Thesis Organization

In chapter 2, we present a detailed review of literature on optical networks. We discuss the components and classifications of optical networks, as well as concepts such as WDM, and static vs. dynamic routing. Some regular and irregular topologies are then described. Of these, the de Bruijn graph and GEMNET are described in detail due to their relevance to this thesis. In chapter 3, we present the three proposed topologies – their connectivity, routing schemes and scaling methods. In chapter 4, we have presented a description of the simulator used to perform the comparisons between the topologies. The most significant simulation results are shown using charts and
tables in chapter 5. We also present a discussion on the experiments including a critical summary on the general observations summarizing the results of all the experiments. Finally, in chapter 6, we have suggested possible topics for future studies in the field.
Chapter 2   Review of Literature

Here, we will discuss in some detail the concept of lightwave technology. Some of the components that go to make up an optical network will be mentioned. This will be followed by a description of single hop and multihop systems, as well as of Wavelength Routed Optical Networks (WRONs). Some concepts such as WDM, and static and dynamic routing will be discussed. We will also discuss some regular and flexible multihop topologies, of which the de Bruijn graph structure and the GEMNET will be discussed in some detail due to their relevance to this thesis.

2.1 Basics of Lightwave Technology

The high capacity of the optical fiber (in the range of Tb/s) has made it highly desirable as a data transmission medium. Optical fibers also have several other advantages such as low signal attenuation (as low as 0.2dB/km), low signal distortion, low power requirement, low material usage, small space requirement, and low cost [25]. Light is transmitted through the fiber using the principle of total internal reflection.

The fiber with the highest data transmission capability is the singlemode fiber. In this, the core is so small that only a single mode is supported. The bandwidth of a single mode fiber far surpasses the capabilities of today’s network electronics. Not only can the fiber support tens of gigabits per second,
it can carry many gigabit channels simultaneously. Single mode is the preferred medium for long distance communications.

2.2 Physical Configurations

Optical fiber networks can be classified into three basic types [29]: the bus, the tree, and the star. This classification is based on the maximum number of users that the configuration can handle.

- **Bus configuration**: In this system, nodes are connected in a sequential order with the bus, and the bus has two sides, the receiving side and the sending side. It can support a maximum of 482 users.

- **Tree configuration**: In this configuration, the maximum number of users cannot be more than 16 because of power splitting. However, with optimal use of amplifiers, it can support a maximum of 512 users.

- **Star configuration**: Here, the transmitters and receivers of the nodes are connected to a star coupler. The excess power loss in this structure is quite small. The maximum number of users can reach 1000.

2.3 Optical components

Many devices constitute the componentry used in today's lightwave technology. In addition to the ones already existing, new ones are constantly being developed [31]. Some of the more common components are described below:
• **LASER:** The laser device is used in optical transmitters. Laser is an acronym for Light Amplification by Simulated Emission of Radiation.

• **Couplers:** A coupler is a device whose job is to divert or direct light into (generally called combiners) or split light out of (generally called splitters) a fiber. Couplers divide signals equally.

• **Wavelength Routers:** These are used to route an input signal to the output port. The wavelength router is a generalization of the wavelength-division multiplexer.

• **Optical Amplifiers:** These devices are used to boost the power of an optical signal.

• **Wavelength-Division Multiplexers and Demultiplexers:** A multiplexer is used to combine different signals from an input port and route them onto an output port. Conversely, a demultiplexer splits a signal and passes or routes the individual wavelengths onto their individual paths.

2.4 **Classifications of Optical Networks**

A local lightwave network can be constructed using WDM and tunable optical transceivers [25]. As mentioned earlier, using WDM, we can divide a high bandwidth optical channel into numerous subchannels operating at peak electronic speeds. A node can transmit data by tuning its transmitters to one of the available wavelength channels. Similarly, another node can receive data by tuning its receivers to receive from the appropriate channel. Based on this
taxonomy, a lightwave network can be classified into three types: single hop networks, multihop networks, and wavelength routed systems.

2.4.1 Single Hop Networks

In a single hop system, a node can transmit data to another node in one hop. This is done entirely in the optical domain, without any conversion into electronic form. Single hop networks are also called all-optical networks [35]. They can be classified into two categories: those requiring pretransmission coordination, and those not requiring any pretransmission coordination [25]. In a pretransmission coordinated system, a single control channel is used to control transmission requirements between stations. Users communicate between each other using data channels. In systems without any pretransmission coordination, no such control channel exists. Arbitration of transmission rights is performed either in a preassigned fashion or through contention-based data transmission on the regular data channels [25].

Figure 1: A star-based single hop system.
2.4.2 Multihop Networks

The multihop network consists of a physically distributed optical topology, and traffic generating and terminating nodes, each of which has been allocated a small number of transmitters and receivers [37]. In a multihop network, a node transmits packets to another node through zero or more intermediate nodes. Each node is assigned a set of channels through which it can send or receive data. These channel assignments are static and their distribution generally do not change unless a change is deemed to be beneficial for the whole network [40]. Unlike single hop systems, data may not be able to be sent between any two pairs of nodes in a single hop. The pattern in which nodes communicate with each other is called logical topology. This pattern simulates the logical order in which nodes communicate with each other. This pattern can be either regular or irregular. The underlying physical connectivity is called the physical topology. The logical and physical topologies of an example four-node multihop network is shown in fig. 2.
Figure 2: Physical and logical topologies of a 4-node multihop network

From the physical topology, it can be seen that node 1 can communicate with node 2 using wavelength $\omega_1$, while it can communicate with node 4 if it sends data to node 2 on $\omega_1$, and then from node 2 to node 4 using $\omega_3$. A wavelength
switch is thus required at node 2. The device at the centre of the network is a passive star coupler used to route the individual wavelengths along their intended paths. The connectivity pattern can be described using a logical topology as shown in figure 2(b). Here, since node 1 can communicate with node 2 using wavelength $\omega_1$, the connection is shown as a directed link between node 1 and node 2, with the wavelength (in this case $\omega_1$) being shown alongside the link.

2.4.3 Wavelength Routed Networks

A wavelength routed system provides a transparent light path between network terminals. A lightpath is an all-optical communication channel between two nodes in the network, and it may constitute of intermediate links. The routing may be done through zero or more intermediate nodes. These nodes route the lightpath using their active switches. Thus these networks do not employ any conversion device. The path of the optical signal is determined by the source user location, by the fiber on which the source user transmits, and by the retransmitted wavelength [16]. The routing operations at the intermediate nodes mainly consist of wavelength conversion and filtering. Since this system does not employ electronic switching devices at the intermediate nodes, congestion so common to multihop networks is greatly avoided.
2.4.4 Static vs. Dynamic Routing Schemes

In wavelength routed networks, wavelength allocation between all possible source-destination pairs of nodes in the network are done by using static and dynamic allocation schemes. In the static scheme, a static end-to-end transparent channel is allocated to every source-destination pair. Hence, for every communication, the path and wavelength are predefined. In [16], a number of topologies are described which use the static routing method. In a dynamic wavelength allocation scheme, WDM channels are assigned to pairs of nodes as needed and then reclaimed when the data transmission is over. In [50], an allocation scheme has been described in which the number of required wavelengths in the network is considerably less than what is needed in the scheme described in [16]. The disadvantage of using static methods is that they assume that each node can communicate simultaneously with every other node and as a result, the number of wavelengths required is large compared to the actual number needed at any given period of time under realistic considerations. Dynamic allocation schemes suffer from the disadvantage that such networks can experience a large time overhead caused by searching for an idle wavelength on all the links that make the path between the source-destination pair.

2.5 Optical Networks based on Regular Topologies

Regular multihop structures are very well suited to the design of optical networks on account of their uniform connection patterns, low hop distance,
and uniform load distribution. In the following sections, some existing regular multihop topologies have been studied. Finally, a brief comparison is made between these topologies.

2.5.1 Kautz graph

The Kautz graph [12] is a directed graph and is often used to design the logical topology of a multihop network. It is represented in the form $K(d, k)$ where $d$ is the degree of the nodes in the graph and $k$ is the diameter of the network. The nodes are labelled with a $k$-length string of the form $(x_1x_2\ldots x_k)$ where for each $x_i$, $1 \leq i \leq k$. There will be a directed edge between two nodes $X\,(x_1x_2\ldots x_k)$ and $Y\,(y_1y_2\ldots y_k)$ if there exists a suffix of length $l$, $1 \leq l < k$, which is also a prefix of $Y$. For example, in fig. 3 shown below, node 10 has a directed edge to 01 while 01 also has a directed edge to 10. The number of nodes $N$ in a Kautz graph $K(d, k)$ is given by $N = d^k + d^{k-1}$.
Figure 3: A K(2, 2) Kautz Digraph

Two routing algorithms for the Kautz digraph are presented in [12]: the shortest path algorithm and the long path algorithm. The shortest path algorithm finds a unique path from the source to the destination; however, it has the disadvantage of sometimes causing congested links, as link loads are not checked during packet transmission. This can cause increased queuing delay. The long path algorithm evenly distributes the packets throughout the network. This can cause increased transmission times, but the traffic is distributed more evenly. In [30], a throughput comparison among Kautz graphs, de Bruijn graphs and ShuffleNets has been done.

2.5.2 ShuffleNet

In the ShuffleNet [17] architecture, the nodes are arranged in rows and columns. Nodes in one column are connected to nodes in adjacent columns by directed links. A \((p, k)\) ShuffleNet can be constructed out of \(N = kp^k\) nodes \((p, k\)
= 1, 2, 3,...) where the nodes are arranged in $k$ columns of $p^k$ nodes each. The $k^{th}$ column can be visualized as wrapped around to the first in a cylindrical manner. The nodes in a column are numbered from top to bottom from 0 to $p^k-1$, and each node $i$ is connected to nodes $j, j+1, \ldots, j+p-1$ in the next adjacent column, where $j = (i \mod p^{k-1}).p$. The diameter of a ShuffleNet is $2k-1$.

Consider fig. 4 shown below. This is a (2, 2) ShuffleNet. Here the nodes in column 1 (0, 1, 2, 3) are connected to the nodes in column 2 (4, 5, 6, 7) with directed links using the connectivity pattern described above. The nodes in column 2 are also similarly connected to the next column, in this case column 1, in a wraparound fashion.

![Diagram of a (2, 2) ShuffleNet](image)

Figure 4: A (2, 2) ShuffleNet.

Routing schemes in ShuffleNets: Three types of routing schemes have been discussed for ShuffleNets; the fixed routing [17], adaptive routing [9], and deflection routing [40] schemes. In fixed routing, each packet contains the
destination information in its header. When it reaches an intermediate node, the address of that node is compared to the destination address. If they are the same, then the processing for that packet is over. If not, the packet is forwarded to another node in the next column. In the adaptive routing scheme, a packet is made to follow a link with the least congestion, even though this link may lead it to follow a longer path to its destination. If more than one such link exists, the packet chooses one at random. In the deflection routing technique, the intermediate node routes (deflects) one of the other packets intentionally along its other lowly congested outgoing link, with the hope that the packet will eventually find its way back to its destination.

2.5.3 Hypercube

The simplest form of the hypercube is the boolean hypercube. This structure has $2^n$ nodes and the nodes can be visualized to be placed at the corners of a $n$-dimensional cube. The diameter of this structure is $\log_2 N$ where $N = 2^n$. The binary representation of any node differs from the binary representation of any of its adjacent nodes by exactly one bit. For example, in fig. 5, node 111 is connected to node 011 and differs in the most significant bit position. In a network, if the total number of nodes in the network can be represented in the form $P^D$, $P$ and $D$ both being integers, then the nodes can be arranged as a $D$-dimensional hypercube and this hypercube will have $P$ nodes in each edge [9]. The average hop distance in a hypercube is given by $h = \left( \frac{(M \log_2 N)}{2(N - 1)} \right)$. 
Two types of hypercube structures, the generalized hypercube (GHC) and the generalized hyperbus (GHB) have been introduced in [9]. In the GHC structure, the total number of nodes is \( N = m_r \times m_{r-1} \times \ldots \times m_1 \) and each processor \( X \) is represented by an \( r \)-tuple \( (x_rx_{r-1}\ldots x_1) \) for \( 0 \leq x_i \leq m_i - 1 \). Each processor is connected to any other processor iff their mixed radix representations as described above differ in exactly one bit. For transmitting a packet, the source node stores the source and destination addresses along with some other information in the packet header. The packet does a digit by digit comparison of the destination address and the address of the node it is currently in. If they match, then the packet has reached it's destination. If not, the packet is sent along the edge represented by the first differing digit in the two addresses. This process continues until the destination is reached.
2.5.4 de Bruijn Graph

The de Bruijn graph [20] has been extensively studied as a model topology for constructing regular optical networks. A de Bruijn graph can be represented as a directed graph $G(V, D)$ [20]. If there are two nodes $a$ ($a_1, a_2, ..., a_D$) and $b$ ($b_1, b_2, ..., b_D$) in the network, then an edge can exist from $a$ to $b$ iff

$$b_i = a_{i+1}; \quad a_i, b_i \in \{0, 1, ..., V - 1\}, \quad 1 \leq i \leq D - 1.$$ 

i.e., if $b_1 = a_2, b_2 = a_3, ..., b_{D-1} = a_D$.

A link between two adjacent nodes $A$ and $B$ may be represented by $(D + 1)$ $V$-ary digits, where the first $D$ digits comprise the address of node $A$ and the last $D$ digits, the address of node $B$. For instance, the edge from 010 to 101 is represented by 0101. When all the nodes in a network are communicating with each other, the maximum hop distance between any two pairs of nodes is called the diameter of the network. In the de Bruijn graph $G(V, D)$, the diameter is $D$. The number of nodes in the network is given by $V^D$.

A de Bruijn graph can be seen as the state transition diagram of a $D$ stage $p$-ary shift register [20]. There is a directed edge from a node $a$ ($a_1, a_2, ..., a_D$) to a node $b$ ($b_1, b_2, ..., b_D$) if $a$ can be obtained from $b$ by left-shifting $a$ by one bit and inserting the leftmost digit of $b$, i.e., if

$$(a_2, a_3, ..., a_D, b_1) = (b_1, b_2, ..., b_D).$$
To find the shortest path from node \( a \) to node \( b \), one needs to obtain the largest suffix of \( a \) which is also a prefix of \( b \). If the length of this common string is \( l \), then the shortest path of \( l \) hops from \( a \) to \( b \) is given by
\[
(a_1a_2...a_{D+l}b_{D+l+1}b_{D+l+2}...b_D)
\]

Consider the \( G(2, 3) \) de Bruijn graph shown in fig. 6. Let \( a = (0, 0, 1) \) and \( b = (1, 0, 1) \). The length \( l \) of the longest common prefix of \( a \) and suffix of \( b \) is 1. Then the shortest path can be represented by \((0, 0, 1, 0, 1)\). Thus the path is \( a = (0,0,1) \rightarrow (0,1,0) \rightarrow (1,0,1) = b \).

![Figure 6: A (2, 3) de Bruijn Graph](image)

The diameter or maximum hop distance in a \( G(V, D) \) de Bruijn graph is equal to \( D \). If \( N = V^D \) is the total number of nodes, then the mean hop distance \( h \) is given by [50]:

19
\[ h \leq D \cdot \left( \frac{N}{N - 1} \right) - \left( \frac{1}{V - 1} \right). \]

In the figure, it can be seen that even though nodes (0,0,0) and (1,1,1) have self loops, these loops carry no traffic. Also, the link from (1,0,0) to (0,0,0) only carries traffic destined for node (0,0,0). Due to these sort of asymmetrical properties of de Bruijn graph, the link load characteristics of the network may be unbalanced. A comparison between the link loadings of a de Bruijn graph and a ShuffleNet [17] (discussed in section 2.5.2) shows that ShuffleNet has lower link loads for the same number of nodes and the same nodal degree. However, one important advantage is that because of the high connectivity of the de Bruijn graph structure, a number of alternate paths exist between any source-destination pair. In [48], another strategy has been proposed called the alternate routing strategy. This strategy states that there are \( V \) paths of maximum length \( D+1 \) in a \( G(V, D) \) graph. Thus, for any source node \( A \) (\( a_1, a_2, \ldots, a_D \)) and any destination node \( B \) (\( b_1, b_2, \ldots, b_D \)), the \( i^{th} \) path, \( 0 \leq i < D \) consists of:

The edge \( a_1a_2\ldots a_D \rightarrow a_2a_3\ldots a_Di \) followed by the shortest path from \( a_2a_3\ldots a_Di \) to \( B \). One of these alternate paths is the shortest path from \( A \) to \( B \).

Example:

Consider a \( G(4, 5) \) de Bruijn graph. Let \( A = 11021 \) and \( B = 21102 \). The alternate paths are given by:

- \( 11021 \rightarrow 10210 \rightarrow 02102 \rightarrow 21021 \rightarrow 10211 \rightarrow 02110 \rightarrow 21102 \)
- \( 11021 \rightarrow 10211 \rightarrow 02110 \rightarrow 21102 \)
- \( 11021 \rightarrow 10212 \rightarrow 02121 \rightarrow 21211 \rightarrow 12110 \rightarrow 21102 \)
11021 → 10213 → 02132 → 21321 → 13211 → 32110 → 21102

Here, path 2 is the shortest path.

2.5.5 Advantages and Disadvantages of Regular Topologies

The regular topologies mentioned above have several advantages. The structure of the networks are symmetrical, and so the average hop distances are relatively small. They also employ simple routing strategies, as a result of which the processing complexity at the nodes is small. Topologies such as de Bruijn graph and ShuffleNet present alternate paths between source-destination node pairs, and so the link load characteristics can be controlled.

In spite of its apparent suitability for constructing large scale dynamic networks, multihop networks also have some disadvantages. Regular structures can only be used in areas where the number of nodes in the network belongs to a fixed set of integers, and it is not possible to scale such a network in arbitrary amounts. These networks are also not suitable for high-throughput, real-time, delay-sensitive traffic. High data rates require loose flow control, which on the other hand gives limited protection against congestion. The regular structures also exhibit low fault tolerance. In addition, multihop networks do not naturally support broadcast and multicast. A quantity of research has been done to overcome these problems [43, 47, 53, 56].
2.6 Flexible Topologies.

Several logical topologies have been proposed which allow the network to be scaled in increments derived from a fixed set of integers. The Hypercube Connected Ring Network (HCRNet) [6] is a modified version of the hypercube structure. It can be scaled in sections of \( n \), where \( n \) is the dimension of the cube. The GEMNET [13] is another scalable topology which is used for the design of optical networks. Due to the relevance to this thesis, we describe the GEMNET architecture in some detail in this section.

2.6.1 GEMNET

GEMNET stands for GEneralized shuffle-exchange Multihop NETwork. It is a generalization of shuffle-exchange networks and it can represent a family of network structures such as ShuffleNet and de Bruijn graph [25]. A GEMNET is represented by the tuple \((K, M, P)\) where \( K \) is the number of rows, \( M \) is the number of nodes per column, and \( P \) is the degree of each node. The nodes are arranged in a cylinder of \( K \) columns such that the nodes are interconnected in the shuffle-exchange pattern similar to ShuffleNet. A GEMNET-patterned network can be scaled arbitrarily. This can be done by adding each new node to the bottom of the columns. In other words, GEMNET can be scaled by creating new rows at the bottom of the network. Each node then has to decide which node it should now connect to based on the new structure, and retune its transceivers accordingly [13].

22
Figure 7: A (2, 5, 2) GEMNET

A (2, 5, 2) GEMNET is shown in fig. 7 [25]. Part (a) shows the physical topology of the network. In this, 10 nodes are connected to each other by means of optical fibers through a passive star. The input and output wavelengths of the transceivers for the nodes are also shown. Part (b) shows
the logical topology for the network. The solid links correspond to the paths with number 0, while the broken lines correspond to the paths with number 1. The \((x, y)\) representation of the nodes signify the node and column numbers, respectively.

**Interconnection Pattern**

In a \((K, M, P)\) GEMNET, the \(N = K \times M\) nodes are arranged in \(K\) columns \((K \geq 1)\) and \(M\) rows \((M \geq 1)\) with each node having degree \(P\). A node is represented by a \((\text{column}, \text{row})\) tuple. A node \(a, 0 \leq a < N\), is located at the intersection of column \(c, 0 \leq c < K\) and row \(r, 0 \leq r < M\), if \(c = (a \mod K)\) and \(r = \lfloor (a/K) \rfloor\), where \(\lfloor \cdot \rfloor\) denotes the largest integer smaller than or equal to \(a/K\). The links emanating from a node are called \(i\)-links. The \(i\)-link from node \((c, r)\) connects it to node \((c', r')\) where \(c' = (c + 1) \mod K\) and \(r' = (r \times P + i) \mod M\), \(0 \leq i < P\).

For example, in fig. 7(b) shown above, node at location \((1,3)\) is connected to nodes at location \(((1+1) \mod 2, (6+1) \mod 5)\), where \(i = 0, 1\), or in other words, to nodes at location \((0, 1)\) and \((0, 2)\).

**Routing Strategy**

When there is only one path from the source to the destination in a GEMNET, the packet follows that path to reach the destination. However, when the value of \(N\) (number of nodes in the network) increases, there may be more than one
shortest path for a particular transmission. Three types of routing algorithms have been described in [13] – unbalanced routing, partially balanced routing and random routing schemes. In the unbalanced routing scheme, the hop distance \( h \) is first found by using the following formula:

Find smallest \( h \) such that \( (M + r_d - (r_s * P)^h) \mod M < P^h \), where \( r_s \) and \( r_d \) are the row numbers of the source and destination nodes, respectively.

The LHS value of the solved equation gives the route code \( R \) for that transmission. If \( R \) can be represented in the form \( (r_1r_2...r_n)_{baseP} \), the node about to send the packet on its \( i^{th} \) hop will route the packet to its \( r_i^{th} \) outgoing link.

In the partially balanced routing scheme, if an alternate path exists, the network assigns a new route code \( R' \) based on the formula

\[
R' = R + ((c_d \mod P) * M) < P^h
\]

provided the RHS of the equation is true. If it is, the new route code is used. In the random routing scheme, the route codes are determined according to the unbalanced routing scheme and a random number is used to choose between these multiple paths. The random routing scheme has been shown to generally perform better than the other two schemes [13], and it is the scheme that we will be using in our simulation.

For example, in the \((2, 5, 2)\) GEMNET shown in fig. 7(b), let the source node \( S = (1, 3) \) and destination node \( D = (1, 2) \). Then we solve the equation
\[(5 + 2 - (3 \times 2^x) \mod 5) \mod 5 < 2^x \text{ to get } x = 2.\]

Applying \(x = 2\) to the LHS of the equation, we get route code \(R = 0\). In base \(P = 2\) notation, this can be represented as \((0,0)\). Thus the path is \(S = (1,3) \rightarrow (0,1) \rightarrow (1,2) = D\).

### 2.6.2 Advantages and Disadvantages of GEMNET

The GEMNET architecture has the advantage of representing both shuffle-exchange and de Bruijn graph based networks, in that a \((K, M, P)\) GEMNET transforms into a ShuffleNet when \(M = P^K\), and into a de Bruijn graph when \(M = P^K\) and \(K = 1\) [25]. In addition, the arbitrarily scalable nature of GEMNET makes it a good candidate for constructing real-time high-volume networks.

The GEMNET structure can only be scaled by adding nodes to the network one column at a time; i.e., the total number of nodes has to be a member of a particular set of integers. This leads to the limitation that GEMNET can only be scaled by one column at any instance of time. For this reason, it has been suggested that the entire GEMNET be designed as one column. The de Bruijn graph based topologies discussed in [34] can be scaled for an arbitrary number of nodes. In addition, for the de Bruijn graph based topology presented in [34], the network has a better throughput than GEMNET with the same number of nodes. Also, to add a node to a GEMNET network, \(O(nd)\) nodes need to have their transmitters and receiver retuned, in comparison to \(O(d)\) retuning in the networks presented in [55, 10, 21].
2.7 Conclusion

This chapter covers some of the regular and flexible topologies that have been investigated in the field of multihop optical networks. It has been found that to design an efficient network, a trade-off has to be made between average hop distance, routing complexity, and link load characteristics. Till now, a network which provides a perfect balance between these metrics has not been found, and perhaps wishing for such a network would be wishful thinking. Scalability of a network is a major concern and lots of research is being conducted into that aspect now. The existing regular multihop architectures do not support arbitrary scaling of the network, thus providing a hindrance to their application as real time dynamic networks. Traffic handling capability is another important aspect. Speed mismatch between electronic switches and optical channels can result in lower processing speeds. A quantity of research has been done in making optical switches, but significant results have not been achieved yet. Thus, while designing an optical network, the physical and economical aspects of the network have to be taken into consideration, and an optimal balance has to be found between them.
Chapter 3  The Proposed Flexible Topologies

3.1  Introduction

Multihop lightwave networks based on regular graphs which have been discussed so far require a major retuning of their transmitters and receivers when the networks have to be scaled. Sometimes, the addresses of the nodes have to be modified. In addition, the network also cannot be scaled for a random number of nodes. In [55, 10, 21], three new topologies have been presented which use the concept of almost regular graphs. These topologies show that the network can be scaled arbitrarily with a low degree of perturbation of the existing link.

3.2  Topology A

In [55], a multihop scalable topology has been described whose graph is in general, not regular. Here, the number of incoming and outgoing edges to a node does not change with the addition or deletion of nodes from the network. The diameter of the proposed topology with \( n \) nodes and maximum outdegree \( d \) is \( O(\log d n) \). The edges to and from the new added nodes can be implemented by defining new lightpaths which are small in number, namely \( O(d) \).
Interconnection Pattern

Let \( k \) be an integer such that \( d^k < n < d^{k+1} \), where \( n \) is the number of nodes in the network. Let \( Z_k \) be the set of all \((k+1)\)-digit strings choosing digits from \( Z = (0, 1, 2, \ldots, d-1) \) and let any string of \( Z_k \) be denoted by \( x_0x_1\ldots x_k \). \( Z_k \) can be divided into \( k+2 \) sets \( S_0, S_1, \ldots, S_{k+1} \) such that all strings having \( x_j \) as the left most occurrence of 0 are included in \( S_i \), and all strings with no occurrence of 0 are included in \( S_{k+1} \). Each string in \( S_i \) is smaller than each string in \( S_j \) if \( i < j \).

For any string \( a = x_0x_1\ldots x_i\ldots x_j\ldots x_k \), the string \( b = x_0x_1\ldots x_j\ldots x_i\ldots x_k \) obtained by interchanging the digits in the \( i^{th} \) and \( j^{th} \) position in \( a \), is called the \( i-j \)-image of \( a \). If a node \( c = y_0y_1\ldots y_k \) exists such that \( c \) can be obtained by left-shifting \( a \) and inserting the leftmost digit from \( c \), then there is a directed edge from \( a \) to \( c \), with the edge number being the leftmost digit of \( c \). There is also a directed edge from a node to it's 0-1-image if that image is a used string in the network. Also, there is an edge of the form \( 0x_1x_2\ldots x_k \rightarrow 0x_2\ldots x_kj \) for all \( j \in Z \) whenever \( x_1 \neq 0 \) and the 0-1-image of the source node is an unused string.

For example, in fig. 8(a) shown below, node 010 has an edge to its 0-1-image 100 while 001 has an edge to 010 obtained by shifting 001 and inserting the digit 1.
Addition of nodes

This section discusses the change in the topology that should occur when a new node is added to the network. Addition of a new node $u$ implies that we will assign the smallest unused string to the newly added node. Let the string be $x0x1...xk \in S_j$. We consider the following three cases:

1. $1 < j \leq k$: for every $v$ given by $x_1x_2...x_t$, $0 \leq t \leq d - 1$, $v \in S_{j-1}$. Therefore $v$ is an used string and we have to add a new edge $u \rightarrow v$ to the network. The node given by $w_0 = 0x_0x_1...x_{k-1}$ is guaranteed to be an used string, since $w_0 \in S_0$ and we have to add a new edge $w_0 \rightarrow u$ to the network.

2. $j = k+1$: If $v = x_1x_2...x_t$, $0 \leq t \leq d - 1$ is an used string, we add a new edge $u \rightarrow v$ to the network. Since $x_1x_2...x_k0 \in S_k$ is an used string, there is at least one $v$ such that $u \rightarrow v$ exists. Similarly, if $w = tx_0x_1...x_{k-1}$, $0 \leq t \leq d - 1$ is an used string, we add a new edge $w \rightarrow u$ to the network. We note that $w_0 = 0x_0x_1...x_{k-1} \in S_0$ is an used string. Therefore, there is at least one $w$ such that $w \rightarrow u$ exists. If $x_k = d - 1$, we delete the edge from $w_0$ to its 0-1-image at this time.

3. $j = 1$: Let $w_c = 0x_0x_2...x_k$ be the 0-1-image of $u$. Before inserting $u$, the node $0x_0x_3...x_k$ was connected to all nodes $v = 0x_2...x_k$, $0 \leq t \leq d - 1$. Thus we have to:

- Delete the edge $w_c \rightarrow v$ for each node $v = 0x_2...x_k$ in the network
- Add an edge $u \rightarrow v$
• Add a new edge $w_0 \rightarrow u$

• If $w_c \neq w_0$, add an edge $w_c \rightarrow u$

Figure 8: Interconnection topology with $d = 2$ for (a) $N = 5$ and (b) $N = 6$

Consider the network with $N = 5$ in fig. 8(a). We choose the smallest unused string $u = 101$ to represent the new node being inserted. The node $u$ will have outgoing edges (shown by solid lines) to all nodes of the form $01j$, to nodes
010 and 011. The 0-1-image of $u$ is 011. Hence all edges from 011 to nodes 010 and 011 are deleted and a new edge from 101 to 011 is inserted. Also a new edge is inserted from 010 to 101. The final network is shown in fig. 8(b).

Routing Strategy

Let $l$ be the length of the longest suffix of the string $S = x_0x_1\ldots x_k$ that is also a prefix of $D = y_0y_1\ldots y_k$ and let $\sigma(S, D)$ denote the string $x_0x_1\ldots x_ky_{l+1}y_{l+2}\ldots y_k$ of length $2(k+1)-l$. If all the $k-l$ substrings of $\sigma(S, D)$ are used strings, then there exists a path of the form

$S = x_0x_1\ldots x_k \rightarrow x_2\ldots x_{2k-1}x_ky_{l+1} \rightarrow \ldots \rightarrow x_ky_1\ldots y_{k-2}y_{k-1} \rightarrow y_0y_1\ldots y_k = D$.

This path is also the shortest path from $S$ to $D$.

However, all the substrings in $\sigma(S, D)$ may not exist. Consider a node $u = x_0x_1\ldots x_k$ whose next node $v = x_1x_2\ldots x_{t+1}$, $0 \leq t \leq d - 1$, does not exist. Then we add an edge from $u$ to its 0-1-image $w = x_20\ldots x_k$. If this is also an unused string, then we add an edge from $u$ to its 0-1-shifted-image, given by $0x_2\ldots x_{t+1}$, $0 \leq t \leq d - 1$.

If a network contains all nodes in $S_0$, $S_1,\ldots$, $S_k$, then there exists an edge $v \rightarrow z = x_1x_2\ldots x_k0$, and the packet will be routed along this edge.

Consider the topology shown in fig 8(b). Suppose $S = 011$ and $D = 001$. Since the only outgoing edge from 011 is to its 0-1-image 101, the first edge of the
path is $011 \rightarrow 101$. From 101, we shift in the successive digits of the destination. So the path is given by:

$$S = 011 \rightarrow 101 \rightarrow 010 \rightarrow 100 \rightarrow 001 = D.$$  

### 3.3 Topology B

Many regular graphs such as de Bruijn, Kautz graph, etc. with $n$ nodes and degree $d$ has a diameter of $O(\log_d n)$ [10]. Hence a topology has to be designed in such a manner that its diameter be as close to $O(\log_d n)$ as possible. In the topology proposed in [55], for $n$ nodes in the network, the indegree and outdegree of the network lies between $d$ and $d+2$, where $n$ and $d$ are any integers with $d \leq n$. This topology is thus almost regular, and the diameter is also $O(\log_d n)$. The edges to and from the new added nodes can be implemented by defining new lightpaths which are small in number, namely $O(d)$.

**Interconnection Pattern**

The interconnection pattern is defined as follows. Let $k$ be the smallest integer such that $d^k < n < (d + 1)^k$. Each node $S$ is represented as a sequence of $d$-based $k$ digits $\sigma = x_k \ldots x_2 x_1$. The nodes are then divided into groups, with their group numbers being determined by the leftmost occurrence of $d$ in $\sigma$. If $n = d^k$, then the interconnection pattern follows the de Bruijn graph. The $i$-conjugate
of $S$ is defined by replacing the leftmost occurrence of $d$ in $S$ by $i$. If a node $K = (y_k \ldots y_2 y_1)$ exists such that $K$ can be obtained by left-shifting $\alpha$ and inserting the leftmost digit from $K$, then there is a directed edge from $S$ to $K$, with the edge number being the leftmost digit of $K$. A node also has a directed edge from itself to its $i^{th}$ conjugate (provided it exists), with the edge number being $i$.

For example, consider the network with $d = 2$ in fig. 9(a) shown below. Here, node $02 \in S_0$ and $11 \in S_2$. There is a directed edge from $11$ to $12$ as $12$ can be obtained by left-shifting $11$ and inserting $2$. Similarly, $12$ has an edge to its $1$-conjugate $11$.

**Addition of Nodes**

This section discusses the change in the topology that should occur when a new node is added to the network. To insert a new node $w$ into the network, the steps that have to be performed are:

- Define the incoming edges to $w$ by replacing some existing edges
- Define the outgoing edges from $w$
- In one case, add one edge.

Nodes are added according to group number, i.e., smallest unused string in a group is assigned to a node. Let the smallest unused string be $w = x_k \ldots x_2 x_1 \in S_j$. Any node $v$ given by $tx_k \ldots x_3 x_2$, such that $0 \leq t \leq d - 1$ must satisfy the
condition \( v \in S_{j+1} \). Before addition of \( w \), \( v \) had directed edges to \( x_1 \)-conjugate of \( v \), i.e., to string \( tx_k \ldots x_{j+1}x_1x_{j-1} \ldots x_2 \). For every \( t \), we now replace edge \( tx_k \ldots x_3x_2 \rightarrow tx_k \ldots x_{j+1}x_1x_{j-1} \ldots x_2 \) by a new edge \( tx_k \ldots x_3x_2 \rightarrow x_k \ldots x_2x_1 \). This defines the incoming edges to \( w \). Each replacement entails deleting one edge and adding a new edge. For all outgoing edges from \( w \), if \( j \neq k \), then \( x_k \ldots x_2x_1t \in S_{j+1} \), for all \( t \ 0 \leq t \leq d \) and is hence an unused string. Therefore, all outgoing edges from \( w \) will be to \( t \)-conjugate of \( w \) for \( 0 \leq t \leq d-1 \), of the form \( x_k \ldots x_2x_1 \rightarrow x_k \ldots x_{j+1}tx_{j-1} \ldots x_1 \).

There is a special case for adding a new edge that does not involve \( w \). If \( x_1 = 0 \), an edge of the form \( tx_k \ldots x_3x_2 \rightarrow tx_k \ldots x_{j+1}0x_1x_{j-1} \ldots x_2 \) has to be added.
Example:

Let us consider the topology shown in fig. 9(a) with $d = 2$ and $N = 6$. Here $k = 2$ and $S_0 = \{00, 01, 10, 11\}$, $S_1 = \{02, 12\}$ and $S_2 = \{20, 21, 22\}$. All the strings of $S_2$ are unused strings and all other strings are used strings. To add a new node to the network, the smallest unused string $20$ has to be added. As it is inserted, the edges $02 \rightarrow 00$ and $12 \rightarrow 10$ are replaced by the edges $02 \rightarrow 20$ and $12 \rightarrow 20$. Also, as discussed above, the outgoing edges from $20$ to $00$, $01$ and $02$ are inserted. This results in the new topology as shown in fig. 9(b).
Routing Strategy

In the routing scheme, the maximum suffix of $S$ which is also a prefix of $K$ is first determined. Let the length of this substring be $l$. Then the $k-l+1$ substrings from the string of length $2k - l$ given by $x_k...x_{2}x_1y_{k-l}...y_2y_1$ define the path from $S$ to $K$, i.e., $S = x_k...x_2x_1 \rightarrow x_{k-1}...x_2x_1y_{k-l} \rightarrow y_k...y_2y_1 = K$. This gives the shortest path from $S$ to $K$. However, if all the nodes represented by the $k-l+1$ substrings do not exist in the network, then the packet has to be routed to the $i^{th}$ conjugate of that node, where $i$ is the current digit that was to be inserted. This $i$-conjugate node is guaranteed to exist within the network as it belongs to a lower group, the nodes of which have already been added. In the special case where an edge from the node to its $d$-conjugate has to be inserted, an edge to its 0-conjugate is inserted instead.

An example routing is now described. Consider the topology with $d = 2$ and $n = 6$ shown in fig. 9(a) [55]. Let $S = 02$ and $D = 11$. Using the regular de Bruijn graph routing technique, the path would be $S = 02 \rightarrow 21 \rightarrow 11 = D$. However, the node 21 does not exist in the network. So from 02, the packet is sent to 00 (the 0-conjugate of 02). Thus the path becomes $S = 02 \rightarrow 00 \rightarrow 01 \rightarrow 11 = D$. The hop distance thus increases by one. In [55], it has been shown that the hop distance remains almost $k$. 

37
3.4 Topology C

In [21], another multihop scalable topology has been described whose graph is not regular. Here, the number of incoming and outgoing edges to a node varies from 1 to \( d + 1 \). The diameter of the proposed topology with \( n \) nodes and maximum outdegree \( d \) is \( O(\log d n) \). The edges to and from the newly added nodes can be implemented by defining new lightpaths which are small in number, namely \( O(d) \).

For example, in fig. 10(a) shown below, node 010 has an edge to its 0-1-image 100 while 000 has an edge to 001 obtained by shifting 000 and inserting the digit 1.

**Interconnection Pattern**

Let \( k \) be an integer such that \( d^k < n < d^{k+1} \), where \( n \) is the number of nodes in the network. Let \( Z_k \) be the set of all \( (k+1) \)-digit strings choosing digits from \( Z = (0, 1, 2, ..., d-1) \) and let any string of \( Z_k \) be denoted by \( x_0 x_1 ... x_k \). \( Z_k \) can be divided into \( k+2 \) sets \( S_0, S_1, ..., S_{k+1} \) such that all strings having \( x_i \) as the left most occurrence of 0 are included in \( S_i \), and all strings with no occurrence of 0 are included in \( S_{k+1} \). Each string in \( S_i \) is smaller than each string in \( S_j \) if \( i < j \).

For any string \( a = x_0 x_1 ... x_i ... x_j ... x_k \), the string \( b = x_0 x_1 ... x_j ... x_i ... x_k \) obtained by interchanging the digits in the \( i^{th} \) and \( j^{th} \) position in \( a \), is called the \( i-j \)-image of \( a \).
If a node $c = y_0y_1...y_k$ exists such that $c$ can be obtained by left-shifting $a$ and inserting the leftmost digit from $c$, then there is a directed edge from $a$ to $c$, with the edge number being the leftmost digit of $c$. There is also a directed edge from a node to its 0-1-image if that image is a used string in the network. Also, there is an edge of the form $0x_1x_2...x_k \rightarrow 0x_2...x_0j$ for all $j \in \mathbb{Z}$ whenever $x_1 \neq 0$ and the 0-1-image of the source node is an unused string.

Addition of Nodes

This section discusses the change in the topology that should occur when a new node is added to the network. Addition of a new node $u$ implies that we will assign the smallest unused string to the newly added node. Let the string be $x_0x_1...x_k \in S_j$. We consider the following two cases:

1. $0 < j \leq k$: let $v = x_1x_2...x_kt$, $0 \leq t \leq d - 1$, then $v \in S_{j-1}$. Therefore $v$ is an used string and we have to add a new edge $u \rightarrow v$ to the network. The node given by $w_0 = 0x_0x_1...x_{k-1}$ is guaranteed to be an used string, since $w_0 \in S_0$. Let $w_c$ be a node in the network such that $u$ is the image of $w_c$ then $w_c \in S_{j-1}$ and is guaranteed to exist in the network. Thus we do the following:
   - Add an edge $u \rightarrow v$
   - Add a new edge $w_0 \rightarrow u$
   - If $w_c \neq w_0$, add an edge $w_c \rightarrow u$

39
2. $j = k+1$: If $v = x_1x_2...x_t$, $0 \leq t \leq d - 1$ is an used string, we add a new edge $u \rightarrow v$ to the network. Since $x_1x_2...x_k0 \in S_k$ is an used string, there is at least one $v$ such that $u \rightarrow v$ exists. Also let $w = t_0x_1...x_{k-1}$, $0 \leq t \leq q - 1$ be an used string. Then we do the following:

- add a new edge $w \rightarrow u$ to the network. We note that $w_0 = 0x_0x_1...x_{k-1} \in S_0$ is an used string. Therefore, there is at least one $w$ such that $w \rightarrow u$ exists.
- Add an edge $u \rightarrow v$ for each $v$ in the network.

![Diagram](image)

(a)

![Diagram](image)

(b)

Figure 10: Interconnection topology with $d = 2$ for (a) $N = 6$ and (b) $N = 7$
Consider the network with $N = 6$ in fig. 10(a). We choose the smallest unused string $u = 110$ to represent the new node being inserted. The following new edges are added to the network.

I. The node $u$ will have outgoing edges (shown by solid lines) to all nodes of the form $10j$, $0 \leq j \leq d-1$, to nodes $100$ and $101$.

II. The node $110$ is the image of $101$. Hence there will be an edge from $101$ to $110$.

III. Finally, there will be an edge from $011$ to $110$.

The final network is shown in fig. 10(b).

**Routing Strategy**

In this section, we present the routing scheme from any source node $S = x_0x_1\ldots x_k \in S_j$ to any destination node $D = y_0y_1\ldots y_k \in S_t$. The routing algorithm is described below:
Find_path

\[(\text{digits\_to\_be\_inserted, pivot\_position}) \leftarrow \text{Find-best-suffix-prefix}(\text{source, dest});\]

current\_node \leftarrow \text{source}

path \leftarrow \text{list consisting of source only}

While (current\_node \neq \text{dest})

\[(\text{current-node, path, pivot\_position}) \leftarrow \text{Insert\_digits}(\text{path, digits\_to\_be\_inserted, pivot\_position});\]

\[(\text{current-node, path, pivot\_position}) \leftarrow \text{Move\_pivot}(\text{path, pivot\_position});\]
end While;

return path;

Now we will briefly explain the three main functions, Find-best-suffix-prefix, Insert_digits and Move_pivot.

The function Find-best-suffix-prefix first find the length \( l \) of the longest set of digits from source string \( S \) which can also be used as digits in destination \( D \).

The ordering of the digits may differ only in the position of the pivot. This function determines the position of this pivot (pivot position) in the source string and the list of new digits that must be shifted in to obtain the destination string.

The function Insert_digits calculates the next node in the path by traversing an edge of the form \( x_0x_1...x_k \rightarrow x_1x_2...x_{kj} \) whenever \( x_1x_2...x_{kj} \) is an used string, or
of the form $0x_1x_2...x_k \rightarrow 0x_2...x_kj$ for all $j \in \mathbb{Z}$. This is done only when the following conditions are satisfied:

a) The destination has not been reached

b) The next node calculated by Insert_digits exists in the network

c) There is at least one more digit to be inserted

d) The left-most digit of current node will not be used as part of the destination string.

The function $Move\_pivot$ calculated the next node by traversing an edge from the current node to it's image. This essentially causes the pivot to move one digit to the right. This may be necessary for one of two reasons:

i. The next node calculated by Insert_digits does not exist, or

ii. It is necessary to move the pivot to its proper position with respect to the destination string $D$.

The next node calculated in this step is used as part of the path only if it exists in the network.

**Example 1:** Consider the topology in fig. 10(a). Let the source node be $S = 001$ and the destination node be $D = 011$. We can find a path from $S$ to $D$ by shifting in successive digits of the destination. The final path is given by $P_1 = 001 \rightarrow 010 \rightarrow 101 \rightarrow 011$. Since each of these nodes exist in the network, $P_1$ is a valid path.
Example 2: Let $S = 011$ and $D = 000$. We first try to reach $D$ by shifting in digits from $D$, thus getting the path $011 \rightarrow 110$. But $110$ is an unused string and the corresponding node does not exist in the network. Thus from $S$ we go to its image $101$. The path from $101$ to $000$ can be constructed by shifting in the digits from $D$. The final path is given by $P_2 = 011 \rightarrow 101 \rightarrow 010 \rightarrow 100 \rightarrow 000$. 
Chapter 4  Simulation Experiments

4.1 Introduction

The purpose of this thesis is to test the performance of the above-described topologies with GEMNET with respect to various performance criteria. Based on these results, the routing strategies of the topologies are to be modified and improved upon until their performance surpasses or at least equals that of GEMNET. In the absence of a real-time physical network, such testing can be achieved only through software simulation.

4.2 Overview of the Topologies

The three topologies and GEMNET have been simulated using the C language. Based on the interconnection pattern and routing scheme, the programs can be customised to produce data for variable numbers of nodes in the network. The topology programs produce three sets of data – the path between all possible source-destination pairs of nodes in the network, the edge numbers between all interconnected nodes in the network, and the physical sequence in which nodes are added to the network. These three sets of data are then passed to a simulator (described in the next section), which processes this data to produce various results. GEMNET has been simulated using the random routing scheme (ref. section 2.6.1).
4.3 Overview of the Simulator

The simulator is designed to be an all-purpose program in which relevant data from the various topologies can be plugged in to produce the required results. It simulates a virtual network with data flow through the links of the network. Various factors which can be expected to be present in a real network, such as bottlenecks and data loss, can be simulated in the program, thus creating a realistic view of an actual network. After factoring all these aspects into the performance of the networks, the simulation data has been collected.

In this study, we start with an initial condition in which all nodes in the network have each generated a communication call to another node in the network. Each communication call is processed, and a number of such calls are discarded so as to give the network a chance to get “up and running”. For all calls being processed, random source-destination pairs are generated, and an attempt is made to transfer a random number of data packets through the specific lightpath. As more and more packets get queued up in the network and the loads on the links increase, more and more attempts will fail. To prevent the network from becoming saturated, failed communication attempts are immediately discarded from the network so as to lighten the loads on the links and intermediate nodal queues. We have chosen to keep on processing communications until an appreciable number of communications have been processed. At this point, we stop the simulation program and collect summary information regarding the performance of the various topologies.
4.4 Details of the program

Main Data Structures: The simulation program has been written in the C language. Due to the large amounts of data being generated by the virtual network because of various network events, linked lists were decided upon as the main data structure which would be used to process the network. The flexible nature of such a list, along with the control which it gave over memory allocation and deallocation, helped in arriving at this decision. Various objects related to the simulation, such as packet generate, packet process and timestamp entities, were simulated using C structures. In addition, temporary data structures such as arrays were used to buffer the data being generated in the course of the simulation, and also to read in data from the topologies supplied in the form of flat files.

Outline of the Algorithm:

1. The topologies: Create the topologies based on interconnection pattern, and use routing algorithms to establish paths between all possible source-destination pairs in the network.

2. Store the results (paths, edge connections, and insertion order of nodes) in flat files.

3. The simulator: Read in input data from flat files.

4. Set all data structures to initial values

5. Initialize network with generate calls for each node being processed.
6. Process network for 500 generate calls; discard data generated from these calls.

7. Process network for approximately 500000 events; record miscellaneous data generated from the run.

8. Repeat steps 4 to 7 for 50 times with different random number values to get a true range for the data.

9. Calculate statistics based on data collected in steps 7 and 8.

10. Repeat steps 1 to 9 for GEMNET.

11. Compare data collected in steps 9 and 10 to get comparisons between GEMNET and topologies.

We now elaborate on some of the simulation steps mentioned above.

### 4.4.1 Simulate Topologies

Here, we have simulated the three topologies using the interconnection patterns and routing algorithms as described in [55, 10, 21]. Since these topologies are modeled on the de Bruijn graph, they are of the form $G = (V, D)$. For topologies 1 and 3, the number of nodes can vary between $V^0$ and $V^{0+1}$, while for topology 2, the number of nodes can vary between $V^0$ and $(V+1)^D$. For simulation purposes, we have chosen the values of both $V$ and $D$ to be 3. This gives us a nodal range of $N = 27$ to $N = 81$ for topologies 1 and 3, and a range of $N = 27$ to $N = 64$ for topology 2.
4.4.2 Store Results

The simulator needs three forms of data from the topologies to simulate their performance. These are:

- Paths for the topologies: These are the lightpaths that can be established for that particular topology using the routing strategies.
- Connecting edges between nodes: These are the edges that are used by a data packet to traverse between any two pairs of adjacent nodes in the network. This information is derived from the interconnection patterns.
- Insertion order of nodes: Since these topologies are scalable, nodes have to be added to the network in a pre-assigned fashion. The order in which the nodes are to be added is needed to determine which nodes are physically present in the network at any given time.

4.4.3 Initialize Parameters and Data Structures

Before actually simulating the performance of the network, the following parameters and data structures are created and initialized:

- Lower and upper limits of network
- Maximum queue length at a node
- Arrays with topology data read in from flat files
- Miscellaneous structures to store network performance data
- A two-dimensional doubly linked list, which is the main structure used to generate and process the events of each communication call.
• Structures for packet processing at individual nodes

### 4.4.4 Process Network

To process the network, we discard an arbitrary number of communications so that the network is in the steady state. Once this is done, the subsequent communications are processed as follows:

• If the event being processed is a Generate call, generate a random destination and number of data packets for that call.

• At each node, perform packet buffering and forwarding. If the event being processed is a Packet Forward call, check to see if the particular communication is still active (i.e., it has not been discarded due to a previous packet loss). If it is still active, forward it to the next node in its path. If next node is destination, collect performance data. If communication has been discarded, discard the packet also.

• Repeat process until terminating condition is reached.

### 4.4.5 Calculate Statistics

Once the simulation is over, various statistics described in section 4.5 are collected.
4.4.6 GEMNET Simulation

The data from GEMNET is also processed using the simulator and similar performance data are collected. This data is then compared to the data from the topologies and a performance comparison is done.

4.5 Statistics to be collected

Here, we describe the various statistics collected during the simulation and also why we felt this information to be important in measuring the performance of a network.

- **Average Hop Distance**: This is the average number of edges a packet has to traverse to get from a source node to a destination node. The lesser the hop distance for an architecture, the more efficient is its routing strategy. For a de Bruijn graph $G(V, D)$, the diameter is $D$.

- **Link Load Distributions**: This is the total load on a link when paths are calculated between all possible source-destination pairs. A large mismatch in these values would indicate that the routing algorithm does not allow for all edges to be traversed uniformly and may cause bottlenecks.

- **Communication Loss**: This is the number of generated communications lost in the network over a period of time expressed as a percentage of the total number of communications processed. The routing algorithms have to be designed so as to keep this value at a minimum.
• **Packet Loss:** This is the number of packets lost in the network over a period of time expressed as a percentage of the total number of packets processed. This value also has to be kept at a minimum in the architecture.

• **Normalised Queuing Delay:** This is the average queuing delay experienced by a packet at an intermediate node between a source-destination pair. Long queuing delays at a node would indicate overloads on some particular links as compared to other links, and would result in large transmission times. Since the length of a buffer at an intermediate node is finite, the routing algorithms should be such that transmission loads are distributed evenly throughout the network.
Chapter 5  Results of Simulation Experiments

Based on simulation results, the following results were obtained from simulating the three topologies. Section 5.1 shows results for topology B (shown as PR2 in the results) in comparison with a GEMNET architecture. In the GEMNET architecture, the nodes have $P = 3$ for $N = 27, 31, 45, 48$, and $P = 4$ for $N = 59, 64$. Results comparing with a GEMNET architecture which has $P = 4$ for $N = 45, 48$ are also shown. Here $P$ is the degree of a node while $N$ is the total number of nodes in the network. We have shown GEMNET with both values of $P$ in the intermediate cases because we wish to show the marked difference in performance that arises when GEMNET has its full complement of $P$ edges, i.e. 4 in this case. Here the values $N = 27, 64$ have been chosen as the architecture resembles a deBruijn graph for those cases, $N = 31, 59$ have been chosen as the architecture slightly differs from a deBruijn graph for those cases, and $N = 45, 48$ have been chosen as these values of $N$ fall in between the upper and lower limits.

In section 5.2, we have shown the results for topology A (shown as PR1 in the results) and topology C (shown as PR3 in the results) in comparison with a GEMNET architecture with $P = 3$ for $N = 27, 55, 81$. 
5.1 Comparison of GEMNET and Topology B

5.1.1 Average Hop Distance

This is the average number of edges traversed by a packet when travelling between a source and a destination.

Table 1: Average Hop Distance for PR2

Figure 11: Average Hop Distance
Here, we see that the average hop distance is comparable to that of GEMNET for \( N = 27, 31, 59 \) and 61, while performance of GEMNET is slightly less in the intermediate cases of \( N = 45 \) and 48. So we conclude that the performance is comparable for average hop distance.

5.1.2 Packet Loss

This is the total number of packets lost per simulation run expressed as a percentage of the total number of packets processed.

Packet Loss Average:

![Packet Loss Mean](image)

Figure 12: Packet loss average for \( N = 27, 31, 45, 48, 59, 64 \).
Packet Loss Standard Deviation:

![Packet Loss Std. Devn.](image)

Figure 13: Packet loss Std. Devn. for \( N = 27, 31, 45, 48, 59, 64 \).

Considering \( P = 3 \) for GEMNET, we see that the packet losses are comparable for \( N = 27, 31, 59 \) and 64, but for the intermediate cases of \( N = 45 \) and 48, GEMNET performs better, with a marked difference being observed for \( N = 48 \). This is because the GEMNET network uses the random routing scheme for this simulation, and this results in a very optimised routing pattern, thus leading to lower queuing delay at the nodes. However, if \( P = 4 \) for GEMNET, then the performance of GEMNET is much better than topology B because GEMNET uses its full complement of \( d+1 \) edges during routing, while a fraction of nodes in topology B have \( d+1 \) edges, while all others have \( d \) edges.
5.1.3 Communication Loss

This is the total number of communications lost per simulation run expressed as a percentage of the total number of communications processed.

Communication Loss Average:

Figure 14: Comm. Loss Average for N = 27, 31, 45, 48, 59, 64.
Communication Loss Standard Deviation:

![Communication Loss Standard Deviation Graph]

Figure 15: Comm. Loss std. devn. for N = 27, 31, 45, 48, 59, 64.

Here also, for P = 3 for GEMNET we see that the communication losses are comparable for N = 27, 31, 59 and 64, but for the intermediate cases of N = 45 and 48, GEMNET performs better, with a marked difference being observed for N = 48. This is again because the GEMNET network uses the random routing scheme for this simulation, resulting in an optimised routing pattern, thus leading to lower queuing delay at the nodes and consequently less data loss. However, if P = 4 for GEMNET, then the performance of GEMNET is much better than topology B because GEMNET uses its full complement of $d+1$ edges during routing, while a fraction of nodes in topology B have $d+1$ edges, while all others have $d$ edges.
5.1.4 Normalized Queuing Delay

This is the average delay experienced by a packet buffered at an intermediate node on its way from a source to a destination.

Queuing Delay Average:

![ Queuing Delay Mean Diagram ]

Figure 16: Queuing delay average for $N = 27, 31, 45, 48, 59, 64$.

Queuing Delay Standard Deviation:

![ Queuing Delay Std. Devn. Diagram ]

Figure 17: Queuing delay std. devn. for $N = 27, 31, 45, 48, 59, 64$.
5.1.5 Link Load Characteristics

This is the total number of edges traversed while establishing paths between all possible source-destination pair of nodes in the network.

<table>
<thead>
<tr>
<th>PR2</th>
<th>GEM</th>
<th>PR2</th>
<th>GEM</th>
<th>PR2</th>
<th>GEM</th>
<th>PR2</th>
<th>GEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>785</td>
<td>819</td>
<td>729</td>
<td>799</td>
<td>712</td>
<td>822</td>
<td>121</td>
</tr>
<tr>
<td>48</td>
<td>1872</td>
<td>1084</td>
<td>1767</td>
<td>1785</td>
<td>1767</td>
<td>1785</td>
<td>1089</td>
</tr>
<tr>
<td>64</td>
<td>2661</td>
<td>2661</td>
<td>2661</td>
<td>2661</td>
<td>2661</td>
<td>2661</td>
<td>2661</td>
</tr>
</tbody>
</table>

Table 2: Link load characteristics for \(N = 27, 31, 45, 48, 59, 64\) and \((P) = 0, 1, 2, 3\).

This table shows the loads on four edges denoted by \(P = 0, 1, 2, 3\) for the two topologies B and GEMNET. \(N\) denotes the number of edges in the network. Here, we see that the loads are evenly distributed on the links for \(P = 0, 1, 2\) in topology B (PR2), while the load on edge \(d+1\) (3 in this case) steadily increases as \(N\) increases. GEMNET, on the other hand, has loads distributed evenly through all its edges for all cases of \(N\). (For \(N = 27, 31\), we have forced GEMNET to have a degree 3). This results in less congestion in GEMNET.
5.2 Comparison of GEMNET with Topologies A & C

In this section, we present the results for topology A and topology C. Since the interconnection pattern of these two are similar, we are presenting the results together in one section. Here, we have only shown three values of $N$ as we have determined that the performance of these topologies with respect to GEMNET is poor and so we have not produced too many simulation data.

5.2.1 Average Hop Distance

<table>
<thead>
<tr>
<th>Nodes</th>
<th>PR1</th>
<th>PR3</th>
<th>GEMNET</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>3.35</td>
<td>2.86</td>
<td>2.48</td>
</tr>
<tr>
<td>81</td>
<td>3.39</td>
<td>3.39</td>
<td>3.39</td>
</tr>
</tbody>
</table>

Table 3: Average Hop Distance for PR1 & PR3

![Average Hop Distance Graph]

Figure 18: Average Hop Distance for PR1 & PR3
Here, we see that GEMNET performs slightly better than topology C (PR3) and topology C performs slightly better than topology A (PR1) for the cases of \(N = 27, 55\) while the performance of all three topologies are equal for \(N = 81\). The difference in performance for \(N = 27\) is due to the fact that topologies A & C encounter unused strings while finding paths, resulting in rerouting of the packets and thus increasing the hop distance. However, for \(N = 81\), all the strings are used strings and direct communication can thus take place.

5.2.2 Packet Loss

Packet Loss Average:

![Packet Loss Mean](Figure 19: Packet Loss Average for \(N = 27, 55, 81\))
Packet Loss Standard Deviation:

![Packet Loss Std. Devn.](image)

**Figure 20: Packet Loss Standard Deviation for N = 27, 55, 81**

We see here that GEMNET performs better than topology A (PR1) and topology C (PR3) for $N = 27, 55$ while the performance is similar for $N = 81$. One reason for this is that the load on edge 0 in topology A is much more than the loads on the other edges (see sec. 5.2.5 below) thus resulting in congestion at the edges. Also, edge $d+1$ in topology C is under-utilized, thus leading to increased loads on the other edges. Secondly, GEMNET utilizes the random routing scheme for its topology, allowing it to choose the most optimal path amongst possibly multiple choices. This allows it to maintain bottlenecks at the edges to a minimum.
5.2.3 Communication Loss:

Comm. Loss Average:

Figure 21: Comm. Loss Average for N = 27, 55, 81

Comm. Loss Standard Deviation:

Figure 22: Comm. Loss Std. Devn. for N = 27, 55, 81
Here also, see that GEMNET performs better than topology A (PR1) and topology C (PR3) for \( N = 27, 55 \) while the performance is similar for \( N = 81 \). Again, the reason for this is that the load on edge 0 in topology A is much more than the loads on the other edges (see sec. 5.2.5 below) thus resulting in congestion at the edges. Also, edge \( d+1 \) in topology C is under-utilised, thus leading to increased loads on the other edges. Secondly, GEMNET utilises the random routing scheme for its topology, allowing it to choose the most optimal path amongst possibly multiple choices. This allows it to maintain bottlenecks at the edges to a minimum.

### 5.2.4 Queuing Delay

**Queuing Delay Mean**

![Graph showing Queuing Delay Mean for N = 27, 55, 81](image)

**Figure 23:** Queuing Delay average for \( N = 27, 55, 81 \)
Figure 24: Queuing Delay std. devn. for N = 27, 55, 81

Here, we see that GEMNET has a lower queuing delay for N = 27, 55 while the delays are similar for N = 81. This is because the load on edge 0 in topology A is much more than the loads on the other edges (see sec. 5.2.5 below) thus resulting in congestion at the edges. Also, edge d+1 in topology C is under-utilised, thus leading to increased loads on the other edges.
5.2.5 Link Load Characteristics

Table 4: Link Load characteristics for N = 27, 55, 81 and edge (P) = 0,1,2,3.

This table shows the loads on four edges denoted by P = 0, 1, 2, 3 for the three topologies A, C, and GEMNET. N denotes the number of edges in the network. In this simulation, we have considered degree P = 3 for GEMNET. Topology A (PR1) has P = 3 while topology C (PR3) has P = 4. From the table, we can see that for all values of N, topology A has increased load on edge 0. This leads to congestion of packets and results in increased data loss and communication loss. Similarly, topology C has less loads on edge d+1 (3 in this case) resulting in under-utilization of the edge. For GEMNET, all edges have the loads evenly distributed amongst them, resulting in lower bottlenecks during data transmission.
4.6 Critical Summary

In the above experiments, we have described the most significant results of our simulation experiments on GEMNET and a network with the graph $G(3,3)$ and $G(3,4)$. We now discuss the conclusions that we arrive after simulating each topology in turn.

**Case 1: Topology A**

In this topology, due to the routing function, the load on edge 0 is much more than that on the other edges, leading to congestion in the network. This in turn leads to increased data loss and queuing delays. As a result, the performance of the network is not comparable to GEMNET and the degraded performance does not justify the lower perturbation in comparison to GEMNET.

**Case 2: Topology B**

For topology 2, we see that in cases where the network resembles a de Bruijn graph, i.e. for $N = 27$ and $N = 64$, its performance is comparable with that of GEMNET. The slight differences in packet and communication losses for $N = 31$ and $N = 59$ are due to the fact that in both cases, all the nodes in GEMNET have their full complement of 4 outgoing edges while in topology 2, only $S_1$ elements have 4 edges while all $S_0$ elements have 3 outgoing edges. Similarly, the slight variation in queuing delay can be attributed to the difference in
insertion order of the nodes (ref. section 4.2). For the intermediate case of \( N = 45 \), topology 2 performs better than GEMNET. However this is due to the reason that we have forced the degree of all the nodes of GEMNET to be 3, while topology B has a mix of nodes with degrees 3 and 4. \( N = 48 \) shows GEMNET again gaining in performance over topology 2, due to the increased number of paths available for data transmission. So we conclude that the degradation of the performance in topology B is acceptable. Since the other advantages such as low retuning of transmitters and receivers and consequently low perturbation of the network are an attractive property of topology B, this architecture is an acceptable topology for designing multihop networks.

However, when we allow all the nodes in GEMNET to have a degree of 4 in the intermediate cases, we see that it performs considerably better than topology B. This is in a sense, unavoidable, as the number of available lowly congested links in GEMNET are much more than that present in topology B. However, this disadvantage can be overcome by making some modifications to the architecture of topology B. Some of these modifications are described in the next chapter.
Case 3: Topology C

In this topology, the interconnection pattern is similar to that of topology A. However, the loads on edge 0 are reduced by distributing that load amongst other edges. This architecture also has the $d+1$ edges compared to topology A's $d$, thus resulting in more evenly distributed loads. However, the routing schema fails to evenly distribute the load among all the edges, resulting in under-utilisation of edge $d+1$. The resulting network performs better than topology A, but still falls short of the mark when compared to GEMNET.
Chapter 6       Future Work and Conclusion

Future Work

As a result of our study, we recommend that some additional work be done in this area. In the case of topology B, we see that the results are comparable. However, these results can be further improved upon by making some modifications to the routing algorithm described in section 3.2. For instance, the common prefix-suffix length is only determined once while finding the path, i.e. during the beginning. However, this could be done at each intermediate stage of the procedure, conceivably resulting in a "smart" algorithm which would keep on trying to find an optimised path at each step. Also, for topology B, a conjugate is determined based on the edge to be traversed. This could result in unequal loads being put on the edges. To rectify this, the algorithm could be modified so that the packet chooses an edge randomly whenever a conjugate has to be determined.

While comparing with the GEMNET topology, we have shown data with GEMNET having it's full complement of edges. The topologies, on the other hand, only have their full quota of edges when their architecture exactly resembles a de Bruijn graph. For all other intermediate cases, the number of edges present in the topologies are less than that of GEMNET. This gives GEMNET an advantage over the topologies. We suggest that the interconnection patterns and routing algorithms be modified so that the nodes
in the network can have as close to a full complement of edges as possible. We strongly believe that this will lead to an improvement in the performance of the networks.

In the case of topology A, we can reduce the edge on edge 0 by having the data packet traversing a random edge whenever a node has to send a packet from itself to its image. In the case of topology C, more packets can be forced through edge \(d+1\), thus reducing the loads on the other edges.

In this thesis, we have only looked at graphs of a specified size, \(G(3, 3)\) and \(G(3, 4)\). We believe that our conclusions will hold for de Bruijn graphs of other sizes as well. However, this remains to be investigated. It is possible that in other sizes, loads over a link may be more evenly distributed, resulting in fewer bottlenecks and consequently a smaller number of data transmission losses.
Conclusion

While doing this thesis work, we attempted to design flexible multihop networks which could be scaled for an arbitrary number of nodes. This was deemed to be especially beneficial in the designing of real-time communication networks. In our approach, we started from a structure resembling a de Bruijn graph, and then attempted to scale the network one node at a time. Each such addition of nodes would result in the retuning of some transmitters and receivers of other nodes to accommodate the new node. Three such topologies were considered, and to measure their performance, we compared them to another flexible multihop topology, GEMNET. The comparison was done based on various performance criteria described previously. Our goal was to prove that the performance of our networks did not degrade appreciably in comparison to GEMNET, thus justifying the $O(d)$ retuning of transmitters and receivers in comparison with the $O(nd)$ retuning of transmitters and receivers in GEMNET.

We implemented a network simulator to model the network operations and performance. Data to the simulator consisted of topology routing and other information which were fed to it from the topology simulation programs. For each set of parameters we studied, we varied source-destination pairs, amount of data transmitted, degrees of the nodes in the network. After a simulation run, we collected various performance-measuring information such as queuing delay, packet loss, etc. to measure the network efficiency.
After investigating the most significant results of our simulation, we found that our second topology was comparable to GEMNET when the architecture resembled a de Bruijn graph, and also in intermediate cases if we restricted the degree of GEMNET. However, GEMNET's performance exceeded that of our topology if GEMNET's degree was set to $d+1$. However, we expect to get better results when our topology is modified to increase the degree of the nodes.

The most significant result of the study is that it is possible to arbitrarily scale a multihop network with retuning of its transmitters and receivers, and the resulting graph is an almost regular graph whose degree varies between $d$ and $d+2$. With modification of the topology routing schemes to include increased edge presence in the nodes, it should be possible to produce a performance comparable to that of other scalable multihop topologies such as GEMNET.
Selected Bibliography


Appendix: Source Code Listings

Here, we are presenting the source code listings of the programs used in our thesis. The entire thesis consists of four 'C' language programs, two self-defined 'C' header files, and one 'C++' language program. The four 'C' programs consist of the three topologies and the simulator. The header files contain functions common to two or more programs. The 'C++' program is used to calculate statistics from the data generated from the simulator. Since it is beyond the scope of this report to present the source code for all the programs, we are only giving the source code for topology B and the simulator.

**Topology B**

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "common.h"

#define NUM_NODES 256

// Function Prototypes

void find_group_list(void);
void sort_group_list(void);
int find_group_no(int, int);
int find_path(int, int, int[]);
ULI change_next_node(ULI, ULI, ULI, int[], int*, ULI);
ULI compute_next_node_id(ULI, int);
ULI find_image(ULI, int, int);
int node_is_in_network(ULI);
void insert_path_into_file(int, int[], int);
void insert_node_edge_structure_into_file(int);
void insert_group_list_into_file(int[]);
int convert(int);
```
// Global Variables

int source, dest;
ULI path_count = 0;
int group_list[400];
int d = 3, k = 3, num_nodes_in_network, num_digits;
int d0_digit_flag = 0, num_nodes_flag = 1, gl_group_flag = 0;
long int matrix[NUM_NODES][5];
int digits_to_be_inserted[20];
int node_edge_structure[NUM_NODES][NUM_NODES];
int link_load[4];

main()
{
    int source_node, dest_node, num_hops,
    total_number_hops, num_communications;
    int path[100];
    int lower_limit, upper_limit, i, j, l;
    float avg_num_hops;
    FILE *fptr;

    num_digits = k;
    lower_limit = 1;
    upper_limit = d;

    // read in number of nodes to be processed

    printf("enter numnodes: ");
    scanf("%d", &num_nodes_in_network);

    // calculate group_list and insert list into data file
    find_group_list();
    insert_group_list_into_file(group_list);

    // initialize misc variables

    for(i = 0; i < 5; i++)
        for(j = 0; j < NUM_NODES; j++)
            matrix[j][i] = 0;

    for(i = 0; i < 4; i++)
        link_load[i] = 0;

    total_number_hops = 0;
    num_communications = 0;

    // process all source-destination pairs

    //
for(i = 0; i < num_nodes_in_network; i++)
    for(j = 0; j < num_nodes_in_network; j++)
    {
        source_node = group_list[i];
        dest_node = group_list[j];

        if(source_node != dest_node)
        {
            num_hops = find_path(source_node, dest_node, path);
            insert_path_into_file(num_hops, path,
                                num_nodes_in_network);
            total_number_hops += num_hops;
            num_communications ++;
        }
    }

    // print collected statistics

    avg_num_hops = (float)total_number_hops/num_communications;

    printf("average number of hops for n = %d is: %.2f",
           num_nodes_in_network, avg_num_hops);

    for(i = 0; i < 4; i++)
    {
        printf("link load for edge %d = %d", i, link_load[i]);
        printf("\n");
    }

    // store routing data into files
    if((fptr = fopen("pr2_path.dat", "a")) == NULL)
        printf("File could not be opened\n");
    else
        fprintf(fptr, "%d ", -999);

    fclose(fptr);

    insert_node_edge_structure_into_file(num_nodes_in_network);

    return 0;
} // end main

// function to determine routing path

int find_path(int source_node, int dest_node, int path[])
{
    ULI source_node_id, dest_node_id, current_node_id, next_node_id;

    // other code
ULI temp_id, temp1_id;
int num_hops = 0, i, k;
int go_on_flag = 1, temp_flag = 0;
int length = num_digits, new_length = num_digits;
int current_node_no, next_node_no, temp;
int index_digit_to_be_inserted = 0;

source_node_id = convert_int_to_node_id(source_node, d+1);
dest_node_id = convert_int_to_node_id(dest_node, d+1);

i = num_digits;

// find length of common prefix-suffix

while((go_on_flag) && (i > 0))
    if(suffix(source_node_id, i, num_digits) ==
        prefix(dest_node_id, i, num_digits))
        go_on_flag = 0;
    else i--;

if (i>0) length = num_digits - i;

current_node_id = source_node_id;
temp_id = dest_node_id;
temp1_id = dest_node_id;

// insert source node as first node in path
path[0] = convert_node_id_to_int(source_node_id, d+1);
path_count++;

// populate digits to be inserted

for (i = length - 1; i >= 0; i--)
{
    digits_to_be_inserted[i] = temp_id & BIT_MASK;
    temp_id = temp_id >> NUM_BITS;
}

// perform actual routing

while(current_node_id != dest_node_id)
{
    // calculate next node by shifting in digit
    next_node_id = (suffix(current_node_id,
        num_digits-1,
        num_digits) << NUM_BITS) |
    digits_to_be_inserted[index_digit_to_be_inserted];
if(node_is_in_network(next_node_id) != 1)
{
    next_node_id = change_next_node(current_node_id, 
        source_node_id, 
        dest_node_id, 
        digits_to_be_inserted, 
        &index_digit_to_be_inserted,temp1_id);

    num_hops++; 
    k = num_digits; 
    go_on_flag = 1;

    //calculate new prefix-suffix length for intermediate node 
    while( (go_on_flag) && (k > 0) ) 
    {
        if(suffix(next_node_id, k, num_digits) == 
            prefix(dest_node_id, k, num_digits))
            go_on_flag = 0;
        else k--;
    } //end while 

    if (k > 0) new_length = num_digits - k;

    //if common prefix-suffix found and it is an improvement, use it 
    if(new_length <= (length - index_digit_to_be_inserted))
    {
        for (i = new_length - 1; i >= 0; i--)
        {
            digits_to_be_inserted[i] = temp1_id & BIT_MASK; 
            temp1_id = temp1_id >> NUM_BITS;
        } //end for
        temp_flag = 1;
    } //end if not-equal

    // advance node 
    current_node_no = convert_node_id_to_int(current_node_id, d+1); 
    next_node_no = convert_node_id_to_int(next_node_id, d+1); 
    temp = digits_to_be_inserted[index_digit_to_be_inserted];

    // populate node-edge structure depending on group number 
    if(d0_digit_flag)
    {
        node_edge_structure[current_node_no][next_node_no] = 0; 
        matrix[current_node_no][0]++; 
        link_load[0]++; 
        d0_digit_flag = 0;
    } 
    else if(g1_group_flag)


```c

node_edge_structure[current_node_no][next_node_no] =
    digits_to_be_inserted[index_digit_to_be_inserted - 1];
if (index_digit_to_be_inserted - 1) < 0)
    matrix[current_node_no][digits_to_be_inserted
        [index_digit_to_be_inserted - 1]]++;
link_load[digits_to_be_inserted[index_digit_to_be_inserted - 1]]++;  
g1_group_flag = 0;
}
else
{
    node_edge_structure[current_node_no][next_node_no] = temp;
    link_load[temp]++;
    matrix[current_node_no][temp]++;
}

if(temp_flag) {
    index_digit_to_be_inserted = 0;
    temp_flag = 0;
}

current_node_id = next_node_id;
path[num_hops] = convert_node_id_to_int(current_node_id, d+1);
path_count++;
}
else
{
    current_node_no = convert_node_id_to_int(current_node_id, d+1);
    next_node_no = convert_node_id_to_int(next_node_id, d+1);
    temp = digits_to_be_inserted[index_digit_to_be_inserted];
    node_edge_structure[current_node_no][next_node_no] = temp;
    link_load[temp]++;
    matrix[current_node_no][temp]++;
    current_node_id = next_node_id;
    index_digit_to_be_inserted++;
    num_hops++;
    path[num_hops] = convert_node_id_to_int(current_node_id, d+1);
    path_count++;
}  // end else
}  //end while

return num_hops;  // return number of edges traversed
}

// function to get image of current node if next node doesn't exist

ULI change_next_node(ULI current_node_id, ULI source_node_id,
ULI dest_node_id,
int digit_to_be_inserted[],
int *index_digit_to_be_inserted, ULI temp_id)
{
    int current_node_no, group_no,i, digit, temp_digit;
    ULI image_of_current_node_id, new_id;
    int src,dst;

    src = convert_node_id_to_int(source_node_id, d+1);
    dst = convert_node_id_to_int(dest_node_id, d+1);

    current_node_no = convert_node_id_to_int(current_node_id, d+1);
    group_no = find_group_no(current_node_no,d+1);
    digit = digit_to_be_inserted[*index_digit_to_be_inserted];
    temp_digit = digit;

    if(digit == d)
    {
        d0_digit_flag = 1;
        temp_digit = 0;
    }

    image_of_current_node_id = find_image(current_node_id, group_no,
                                             temp_digit);
    if(node_is_in_network(image_of_current_node_id))
    {
        if( (group_no == 1) && (digit != d) )
        g1_group_flag = 1;

        return image_of_current_node_id;
    }
    else
    {
        new_id = compute_next_node_id(current_node_id,
                                        digit_to_be_inserted[*index_digit_to_be_inserted]);
        (*index_digit_to_be_inserted)++;
        return new_id;
    }
} //end func

// function to check whether node being processed in in network

int node_is_in_network(ULI next_node_id)
{
    int node_no, i;

    87
node_no = convert_node_id_to_int(next_node_id,d+1);

for(i = 0; i < num_nodes_in_network; i++)
    if(group_list[i] == node_no)
        return 1;

return 0;

// function to find the image of a node
ULI find_image(ULI current_node_id, int group_no, int digit)
{
    ULI temp1, temp2, temp;
    int i, total = 0;

    if(group_no == 1)
    {
        current_node_id = current_node_id >> NUM_BITS;
        current_node_id = current_node_id << NUM_BITS;
        current_node_id += digit;
        return current_node_id;
    }

    temp1 = suffix(current_node_id, group_no - 1, k);
    for(i = 1; i <= (k - group_no); i++)
    {
        temp2 = (suffix(prefix(current_node_id, i, k), 1, k))
            << (k-i)*NUM_BITS;
        total += temp2;
    }

    digit = digit << ((group_no - 1)*NUM_BITS);
    temp = temp1 + digit + total;

    return temp;
}

ULI compute_next_node_id(ULI current_node_id, int digit_to_be_inserted)
{
    ULI next_node_id, temp1;

    temp1 = suffix(current_node_id, k-1, k+1) << NUM_BITS;

    next_node_id = temp1 + (ULI)digit_to_be_inserted;

    return next_node_id;
// function to find the group list
void find_group_list(void)
{
    int group_no[MAXNODE], gr_no;
    int group_list_index = 0;
    int maximum_nodes = d+1, group_number, index, i, j;

    for(i = 1; i<k; i++)
        maximum_nodes *= d+1;

    // initialize node edge structure
    for(i = 0; i < NUM_NODES; i++)
        for(j = 0; j < NUM_NODES; j++)
            node_edge_structure[i][j] = -999;

    for(i = 0; i < maximum_nodes; i++)
    {
        gr_no = find_group_no(i,d+1);
        group_no[i] = gr_no;
    }

    for(group_number = 0; group_number <= k; group_number++)
        for(index = 0; index < maximum_nodes; index++)
            if(group_no[index] == group_number)
            {
                group_list[group_list_index] = index;
                group_list_index++;
            }
    group_list[index] = -999;
}

// function to calculate group number of a node
int find_group_no(int node_no,int m)
{
    ULI node_id,temp;
    int position, temp1;

    node_id = convert_int_to_node_id(node_no,m);

    for(position = 1; position <= k; position++)
    {
        temp = suffix(prefix(node_id,position,k),1,k);

        // code continues here...
    }

    return group_no[index];
}
if(temp == d)
{
    temp1 = (k+1) - position;
    return temp1;
}

return 0;
} // end func

// function to insert node edge structure into file
void insert_node_edge_structure_into_file(int num_nodes)
{
    FILE *fptr;
    int i, j, maxnum_nodes = d+1;

    for(i = 1; i < k; i++)
        maxnum_nodes *= d+1;

    if((fptr = fopen("pr2_edge.dat", "a+")) == NULL) // if failure to open file, err
        printf("File could not be opened\n");
    else
    {
        for(i = 0; i < maxnum_nodes; i++)
        {
            for(j = 0; j < maxnum_nodes; j++)
                fprintf(fptr, "%d ", node_edge_structure[i][j]);
            fprintf(fptr, "\n");
        }
    }

    fclose(fptr); //close file after insertion done
} //end func

// function to insert path into file
void insert_path_into_file(int num_hops, int path[], int num_nodes)
{
    FILE *fptr;
    int i;

    if((fptr = fopen("pr2_path.dat", "a+")) == NULL) // if failure to open file, err
        printf("File could not be opened\n");
    else
    {
        if(num_nodes_flag)
        {
            fprintf(fptr, "%d \n", num_nodes);

70
num_nodes_flag = 0;
}
for(i = 0; i < (num_hops + 1); i++)
    fprintf(fptr, "%d ", path[i]);
    fprintf(fptr, "\n");
}  //end else

fclose(fptr);  //close file after insertion done
}  //end func

// function to insert group list into file
void insert_group_list_into_file(int group_list[])
{
    FILE *fptr;
    int i;

    if((fptr = fopen("pr2_glist.dat", "a+")) == NULL)  // if failure to open file, err
        printf("File could not be opened\n");
    else
        {  
            for(i = 0; i < num_nodes_in_network; i++)
                fprintf(fptr, "%d ", group_list[i]);
                fprintf(fptr, "\n");
            }  // end else

    fclose(fptr);  //close file after insertion done
}  // end func

Simulator

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<math.h>
#include "random.h"

#define TOTAL_NODE_DIAM 100
#define LENGTH_OF_PATH_ARRAY 400000
#define MAX_QUEUE_LENGTH 300
#define NUM_NODES 256
#define NUM_EDGES 5
#define INSERT 2
#define GENERATE 1
#define FORWARD 0
#define MAX_NUM_EDGES 4
#define NUM_ROWS 3
#define NUM_COLUMNS 3
#define NUM_CELLS 500000

struct time_node    // structure for timestamps
{
    struct time_node *timenode_next;
    int timestamp;
    void *event_ptr;
};
typedef struct time_node *timenode_ptr;

struct event_node_forward // structure for FORWARD events
{
    int type_of_work;
    void *event_next;
    int comm_num;
    int* path_list_ptr;
    int total_num_packets;
    int start_timestamp;
    int num_packets_remaining;
};
typedef struct event_node_forward *eventnodeforward_ptr;

struct event_node_insert // structure for INSERT events
{
    int type_of_work;
    void *event_next;
    int comm_num;
    int* path_list_ptr;
    int total_num_packets;
    int start_timestamp;
    int num_packets_remaining;
};
typedef struct event_node_insert *eventnodeinsert_ptr;

struct event_node_generate // structure for GENERATE events
{
    int type_of_work;
    void *event_next;
    int comm_num;
    int source_node;
};
typedef struct event_node_generate *eventnodegenerate_ptr;

int path_array[LENGTH_OF_PATH_ARRAY];
int last_index_used = 0, num_nodes_in_network;
int* ptr_array[NUM_NODES][NUM_NODES];

// global variables

int node_edge_weight_structure[NUM_NODES][4];
int node_queue_length_structure[NUM_NODES][NUM_EDGES];
int grouplist_array[NUM_NODES];
int node_edge_structure[NUM_NODES][NUM_NODES];
unsigned int transmission_lost_structure[NUM_CELLS];
int node_edge_lost_structure[NUM_NODES][MAX_NUM_EDGES];
int src_dst_gen_structure[NUM_NODES][NUM_NODES];

int total_num_packets = 0, total_time = 0, TS_count = 0,
lost_comm_count = 0, total_num_comms = 0;
timenode_ptr timenode_first = NULL;
int MaxNodeNumberPlusOne, TotalNumNodes;
char file_path_name[30], file_edge_name[30], file_group_list_name[30];

static long table[32];
long int first_time, Z = 0, rprev = 0;
int comm_num = 1;
unsigned long int val;
long int total_number_of_packets = 0, total_lost_packets = 0;

/* FUNCTION PROTOTYPES */

void process_network(void);
int check_if_comm_is_discarded(void*);
void populate_structures_from_file(void);
void initialise_path_pointers(void);
void find_path(int, int);
void insert_event_into_DS(timenode_ptr*, void*, int, int);
timenode_ptr find_the_time_node(timenode_ptr, int);
void* retrieve_next_task(timenode_ptr*, int*);
void initialise(void);
void process_task(void*, int);
int get_forwarding_time(int, int, int, int);
int is_in_network(int);
void calculate_stats(void);
int get_time_for_next_call(void);
void update_queue(int, int);

main(int argc, char * argv[])
{
    long int i = 0, j;

    // read in data input file names

    strcpy(file_path_name, argv[1]);
    strcpy(file_edge_name, argv[2]);
    strcpy(file_group_list_name, argv[3]);
    sscanf(argv[4], &TotalNumNodes);

    populate_structures_from_file();
    initialise_path_pointers();
    printf("enter rand value key\n");
    scanf("%d", &val);
    init_rand_gen();
    initialise();

    while( (i++ < 500000) && (timenode_first ! = NULL) )
        process_network();

    calculate_stats();

93
return 0;
}

// function to process the events of the network
void process_network(void)
{
    void *task_ptr;
    int time = 0;
    timenode_ptr temp1;

    task_ptr = retrieve_next_task(&timenode_first, &time);
    if (task_ptr == NULL)
    {
        printf("DONE
");
        return;
    }

    process_task(task_ptr, time);
}

// function to retrieve next task for processing

void* retrieve_next_task(timenode_ptr *ptr_start_node, int* time)
{
    void* temp;
    eventnode_degenerate_ptr temp1;
    eventnodeinsert_ptr temp2;
    eventnodeforward_ptr temp3;

    if (*ptr_start_node == NULL)
    return NULL;

    temp = (*ptr_start_node) -> event_ptr;
    *time = (*ptr_start_node) -> timestamp;

    temp1 = (eventnode_degenerate_ptr) temp;
    temp2 = (eventnodeinsert_ptr) temp;
    temp3 = (eventnodeforward_ptr) temp;

    if (((temp1 -> type_of_work) == GENERATE)
        (*ptr_start_node) -> event_ptr
        = temp1 -> event_next;
    else if (((temp2 -> type_of_work) == INSERT)
{  
        if (temp2 -> num_packets_remaining == 0)
            (*ptr_start_node) -> event_ptr
            = temp2 -> event_next;
        temp2 -> num_packets_remaining --;
    }
    else if (((temp3 -> type_of_work) == FORWARD)
        (*ptr_start_node) -> event_ptr
        = temp3 -> event_next;

    return temp;
}
} //end func

// function to process retrieved task

void process_task(void *task_ptr, int time)
{
    int num_packets, source_node_no, destination_node_no, j,
        time_for_next_call, temp_time, time_for_forward_event;
    int num_packets_remaining, next_node_num, temp_node_no;
    int outgoing_edge_num, elapsed_time;
    int time_when_packet_will_be_processed, next_to_next_node_num;
    timenode_ptr temp1;
    eventnodeforward_ptr temp3, ptr_to_forward_event;
    eventnodegenerate_ptr temp, ptr_to_generate_event;
    eventnodeinsert_ptr temp2, ptr_to_insert_event;
    int *temp_ptr;

    temp = (eventnodegenerate_ptr) task_ptr;
    temp2 = (eventnodeinsert_ptr) task_ptr;
    temp3 = (eventnodeforward_ptr) task_ptr;

    if((temp -> type_of_work) == GENERATE)
    {
        source_node_no = temp -> source_node;
        destination_node_no = randgen() % MaxNodeNumberPlusOne;
        while ( ( !(is_in_network(destination_node_no)) ||
                (source_node_no == destination_node_no))
            destination_node_no = randgen() % MaxNodeNumberPlusOne;

        temp_ptr = ptr_array[source_node_no][destination_node_no];
        next_node_num = *(temp_ptr + 1);
        outgoing_edge_num =
            node_edge_structure[source_node_no][next_node_num];

        if(transmission_lost_structure[temp -> comm_num] == 1)
            goto label1;

        src_dst_gen_structure[source_node_no][destination_node_no]++;

        if(node_queue_length_structure[source_node_no][outgoing_edge_num]
            > MAX_QUEUE_LENGTH)
        {
            transmission_lost_structure[temp -> comm_num] = 1;
            node_edge_lost_structure[source_node_no][outgoing_edge_num]++;
            lost_comm_count ++;
            goto label1;
        } // end if

    } // generate random number of packets for comm.
    num_packets = randgen() % 65;
    while(num_packets == 0)
    {
        num_packets = randgen() % 65;
        total_number_of_packets += num_packets;
    }
    num_packets_remaining = num_packets - 1;

    95
node_queue_length_structure[source_node_no][outgoing_edge_num]++;

// create inset event for data structure & store it

ptr_to_insert_event = malloc(sizeof(struct event_node_insert));
ptr_to_insert_event -> type_of_work = INSERT;
ptr_to_insert_event -> comm_num = temp -> comm_num;
ptr_to_insert_event -> path_list_ptr =
ptr_array[source_node_no][destination_node_no];
ptr_to_insert_event -> total_num_packets = num_packets;
ptr_to_insert_event -> num_packets_remaining = num_packets_remaining;
ptr_to_insert_event -> start_timestamp = time + 1;
insert_event_into_DS(&timenode_first,
(void *)ptr_to_insert_event,
time + 1, INSERT);

time_for_next_call = time + 1 + get_time_for_next_call();

// create next generate call for that node & insert it

ptr_to_generate_event = malloc(sizeof(struct event_node_generate));
ptr_to_generate_event -> type_of_work = GENERATE;
ptr_to_generate_event -> comm_num = comm_num;
ptr_to_generate_event -> source_node = source_node_no;
insert_event_into_DS(&timenode_first,
(void *)ptr_to_generate_event,
time_for_next_call,
GENERATE);

comm_num ++;
}

// end if-insert

else if((temp3 -> type_of_work) == FORWARD)
{
    temp_node_no = *(temp3 -> path_list_ptr);
next_node_num = *((temp3 -> path_list_ptr) + 1);
next_to_next_node_num = *((temp3 -> path_list_ptr) + 2);

if(transmission_lost_structure[temp3 -> comm_num] == 1)
{
    update_queue(temp_node_no,
        node_edge_structure[temp_node_no][next_node_num]);
total_lost_packets++;
goto label1;
}

if(next_to_next_node_num != -1)
outgoing_edge_num =
        node_edge_structure[next_node_num][next_to_next_node_num];

// if destination arrived
if(next_to_next_node_num == -1)
{
    update_queue(temp_node_no,
        node_edge_structure[temp_node_no][next_node_num]);
if(temp3 -> num_packets_remaining == 0)
{
elapsed_time = time - (temp3 -> start_timestamp);
total_time += elapsed_time;
total_num_packets += temp3 -> total_num_packets;
timenode_first -> event_ptr = temp3 -> event_next;
free(temp3);
}
else
{
timenode_first -> event_ptr = temp3 -> event_next;
free(temp3);
}
else
{
(temp3 -> path_list_ptr)++;

if(node_queue_length_structure[next_node_num][outgoing_edge_num] > MAX_QUEUE_LENGTH)
{
    transmission_lost_structure[temp3 -> comm_num] = 1;
    update_queue(temp_node_no, outgoing_edge_num);
    total_lost_packets++;
    node_edge_lost_structure[next_node_num][outgoing_edge_num]++;
    lost_comm_count ++;
    goto label1;
}
else
    node_queue_length_structure[next_node_num][outgoing_edge_num] ++;

update_queue(temp_node_no, outgoing_edge_num);

temp_time =
    node_edge_weight_structure[next_node_num][outgoing_edge_num];

time_when_packet_will_be_processed =
    get_forwarding_time(outgoing_edge_num, next_node_num, temp_time, time);
insert_event_into_DS(&timenode_first,
    (void *)&temp3,
    time_when_packet_will_be_processed,
    FORWARD);
}
} // end if-forward
else if((temp2 -> type_of_work) == INSERT)
{
    temp_node_no = *(temp2 -> path_list_ptr);
    next_node_num = *((temp2 -> path_list_ptr) + 1);
    outgoing_edge_num =
        node_edge_structure[temp_node_no][next_node_num];
if(transmission_lost_structure[temp2 -> comm_num] == 1)
{
    update_queue(temp_node_no, outgoing_edge_num);
    total_lost_packets++;
    goto label1;
}
if(*(temp2 -> path_list_ptr) + 2) == -1)
{
    update_queue(temp_node_no, outgoing_edge_num);
    if(temp2 -> num_packets_remaining == 0)
    {
        elapsed_time = time - (temp2 -> start_timestamp);
        total_time += elapsed_time;
        total_num_packets += temp2 -> total_num_packets;
        timenode_first -> event_ptr = temp2 -> event_next;
        free(temp2);
    }
    else
    {
        timenode_first -> event_ptr = temp2 -> event_next;
        free(temp2);
    }
}
else
{
    if(node_queue_length_structure[next_node_num][outgoing_edge_num] > MAX_QUEUE_LENGTH)
    {
        update_queue(temp_node_no, outgoing_edge_num);
        total_lost_packets++;
        transmission_lost_structure[temp2 -> comm_num] = 1;
        node_edge_lost_structure[next_node_num][outgoing_edge_num]++;
        lost_comm_count++;
        goto label1;
    }
    else
    {
        node_queue_length_structure[next_node_num][outgoing_edge_num]++;
        update_queue(temp_node_no, outgoing_edge_num);

        ptr_to_forward_event =
        malloc(sizeof(struct event_node_forward));
        ptr_to_forward_event -> type_of_work = FORWARD;
        ptr_to_forward_event -> comm_num = temp2 -> comm_num;
        ptr_to_forward_event -> path_list_ptr = (temp2 -> path_list_ptr) + 1;
        ptr_to_forward_event -> total_num_packets =
        temp2 -> total_num_packets;
        ptr_to_forward_event -> num_packets_remaining =
        (temp2 -> num_packets_remaining) + 1;
        ptr_to_forward_event -> start_timestamp = temp2 -> start_timestamp;

        temp_time = node_edge_weight_structure[temp_node_no][outgoing_edge_num];

        time_for_forward_event =
        get_forwarding_time(outgoing_edge_num, temp_node_no, temp_time, time);

        insert_event_into_DS(&timenode_first,
                             (void *)ptr_to_forward_event,
                             time_for_forward_event,
                             FORWARD);

    }
}

98
if((timenode_first -> event_ptr) == NULL)
{
    temp1 = timenode_first;
    timenode_first = temp1 -> timenode_next;
    free(temp1);
    TS_count++;
}
} // end-func

// function to determine that queue lengths do not get abnormal values

void update_queue(int node_no, int edge_num)
{
    node_queue_length_structure[node_no][edge_num] -=
    if(node_queue_length_structure[node_no][edge_num] < 0)
        node_queue_length_structure[node_no][edge_num] = 0;
} // end func

// function to get forwarding timestamps based on queue lengths

int get_forwarding_time(int edge_num, int node_no, int temp_time, int time)
{
    int time_when_packet_will_be_processed;

    if(temp_time < time)
    {
        time_when_packet_will_be_processed = time + 1;
        node_edge_weight_structure[node_no][edge_num] =
        time_when_packet_will_be_processed + 1;
    }
    else
    if(temp_time == time)
    {
        time_when_packet_will_be_processed = time + 1;
        node_edge_weight_structure[node_no][edge_num] =
        time_when_packet_will_be_processed + 1;
    }
    else
    if(temp_time > time)
    {
        time_when_packet_will_be_processed = temp_time;
        node_edge_weight_structure[node_no][edge_num]++;
    }

    return time_when_packet_will_be_processed;
} // end func

// function to insert events into main data structure

void insert_event_into_DS(timenode_ptr *ptr_start_ptr,
                        void* new_event_ptr, int time, int event_type)
{
tinenode_ptr start_ptr, temp, current_ptr;
eventnodeinsert_ptr ptr_to_insert_event;
eventnodeforward_ptr ptr_to_forward_event;
eventnodegenerate_ptr ptr_to_generate_event;

start_ptr = ptr_start_ptr;
current_ptr = find_the_time_node(start_ptr, time);
temp = current_ptr -> event_ptr;

if (event_type == INSERT)
{
    ptr_to_insert_event = (eventnodeinsert_ptr) new_event_ptr;
    ptr_to_insert_event -> event_next = temp;
}

else if (event_type == FORWARD)
{
    ptr_to_forward_event = (eventnodeforward_ptr) new_event_ptr;
    ptr_to_forward_event -> event_next = temp;
}

else if (event_type == GENERATE)
{
    if(start_ptr == NULL)
    {
        *ptr_start_ptr = current_ptr;
        ptr_to_generate_event = (eventnodegenerate_ptr) new_event_ptr;
        ptr_to_generate_event -> event_next = temp;
        total_num_comms ++;
    }

current_ptr -> event_ptr = new_event_ptr;
} //end func

// function to determine correct insertion timestamps for events
timenode_ptr find_the_time_node(timenode_ptr start_ptr, int time)
{
    timenode_ptr current_ptr, prev_ptr, temp;

    if(start_ptr == NULL)
    {
        current_ptr = malloc(sizeof(struct time_node));
        current_ptr->timestamp = time;
        current_ptr -> event_ptr = NULL;
        current_ptr -> timenode_next = NULL;
        return current_ptr;
    }

    while(start_ptr -> timestamp <= time)
    {
        prev_ptr = start_ptr;
        if((start_ptr -> timenode_next == NULL) || (start_ptr -> timestamp == time))
            break;
        start_ptr = start_ptr -> timenode_next;
    }
if (start_ptr -> timestamp == time)
    current_ptr = start_ptr;
else if (start_ptr -> timestamp > time)
{
    temp = malloc(sizeof(struct time_node));
    temp -> timenode_next = start_ptr;
    temp -> timestamp = time;
    temp -> event_ptr = NULL;
    prev_ptr -> timenode_next = temp;
    return temp;
}
else
{
    current_ptr = start_ptr -> timenode_next;
    if(current_ptr == NULL)
{
        current_ptr = malloc(sizeof(struct time_node));
        start_ptr -> timenode_next = current_ptr;
        current_ptr->timestamp = time;
        current_ptr -> event_ptr = NULL;
        current_ptr -> timenode_next = NULL;
    }
}
return current_ptr;
} // end func

// function to initialise misc. variables for the simulation

void initialise(void)
{
    int index1, selected_node,
    remaining_number_nodes = num_nodes_in_network, time = 1;
    long int index2;
    eventnodegenerate_ptr ptr_to_generate_event;

    for(index1 = 0; index1 < NUM_NODES; index1++)
    for(index2 = 0; index2 < MAX_NUM_EDGES; index2++)
        node_edge_lost_structure[index1][index2] = 0;

    for(index1 = 0; index1 < NUM_NODES; index1++)
    for(index2 = 0; index2 < MAX_NUM_EDGES; index2++)
        src_dst_gen_structure[index1][index2] = 0;

    /* Initialise array used to store whether a transmission is lost or not */
    for(index1 = 0; index1 < NUM.Cells; index1++)
        transmission_lost_structure[index1] = 0;

    /* Initialise array used to store the queue length at each node */
    for(index1 = 0; index1 < NUM_NODES; index1++)
    for(index2 = 0; index2 < NUM.Edges; index2++)
        node_queue_length_structure[index1][index2] = 0;
/* CODE SEGMENT TO STORE EACH NODE IN NETWORK WITH A GENERATE CALL */

for (index1 = 0; index1 < num_nodes_in_network; index1++)
{
    if(remaining_number_nodes == 0)
        break;
    else {
        index2 = randgen() % remaining_number_nodes;
        selected_node = grouplist_array[index2];

        ptr_to_generate_event = malloc(sizeof(struct event_node_generate));
        ptr_to_generate_event -> type_of_work = GENERATE;
        ptr_to_generate_event -> comm_num = comm_num;
        ptr_to_generate_event -> source_node = selected_node;
        insert_event_into_DS(&timenode_first,
            (void *)ptr_to_generate_event,
            time, GENERATE);

        time ++;
        comm_num ++;
        grouplist_array[index2] =
            grouplist_array[remaining_number_nodes - 1];
        remaining_number_nodes--;
    }
} // end else
} // end func

// function to check if node is in network

int is_in_network(int node)
{
    int index = 0;

    while(path_array[index] != -999)
    {
        if(path_array[index] == node)
            return 1;
        else
            index++;
    } return 0;
} // end func

// function to calculate statistics after each run

void calculate_stats(void)
{
    float avg_time;

    avg_time = (float)(total_time) / total_num_packets;

    printf("Average packet delay for N = %d is %.2f\n",......
printf("Num Comms lost in %d time units = \%d\n", TS_count, lost_comm_count);
printf("Total num comms = %d\n", total_num_comms);
printf("percentage of lost comm. = %.2f\n",

(float)lost_comm_count/total_num_comms);
printf("lost = %d, total = %d\n", total_lost_packets, total_number_of_packets);
printf("percentage of lost packets = %.2f\n",

(float)total_lost_packets/total_number_of_packets);
return;
} } // end func

// function to determine paths between pairs of nodes

void initialise_path_pointers(void)
{
    long int index = 0, temp_index;
    int source_node, dest_node;

    ptr_array[0][1] = &path_array[index];

    while(path_array[index] != -1)
        index++;
    index++;

    do
    {
        source_node = path_array[index];
        temp_index = index;
        while(path_array[index] != -1)
            index++;
        dest_node = path_array[index - 1];
        ptr_array[source_node][dest_node] = &path_array[temp_index];
        index++;
    } while(path_array[index] != -999);
} } // end func

// function to read in topology data from files

void populate_structures_from_file(void)
{
    FILE *path_fptr, *edge_fptr, *grlist_fptr;
    int j, temp, max = -999;
    long int i;

    if((path_fptr = fopen(file_path_name, "r")) == NULL)
        printf("File could not be opened\n");
    else
    {
        fscanf(path_fptr, "%d", &num_nodes_in_network);
        for(i = 0; i < LENGTH_OF_PATH_ARRAY; i++)
            { 
                fscanf(path_fptr, "%d", &path_array[i]);
                if(max < path_array[i])
                    max = path_array[i];
                if(path_array[i] == -999)
                    103
{ MaxNodeNumberPlusOne = max + 1; break; }

if((edge_fptr = fopen(file_edge_name, "r")) == NULL)
    printf("File could not be opened\n");
else
    for(i = 0; i < TotalNumNodes; i++)
        for(j = 0; j < TotalNumNodes; j++)
            fscanf(edge_fptr,"%d", &node_edge_structure[i][j]);

if((grlist_fptr = fopen(file_group_list_name, "r")) == NULL)
    printf("File could not be opened\n");
else
    for(i = 0; i < num_nodes_in_network; i++)
        fscanf(grlist_fptr,"%d", &grouplist_array[i]);

close(path_fptr);
close(edge_fptr);
close(grlist_fptr);

// function to get time for next generate call
int get_time_for_next_call(void)
{
    float u, ret_val;
    int val;

    u = (float)(randgen) % 65537;
    u = u/65537;

    ret_val = log(1 - u) * (-100);
    val = (int)ret_val;
    return val;
} // end func

// function to check if communication being processed is discarded
int check_if_comm_is_discarded(void *task_ptr)
{
    eventnodeforward_ptr temp3;
    eventnodegenerate_ptr temp1;
    eventnodeinsert_ptr temp2;
    int comm_num;

    temp1 = (eventnodegenerate_ptr) task_ptr;
    temp2 = (eventnodeinsert_ptr) task_ptr;
    temp3 = (eventnodeforward_ptr) task_ptr;

    if(temp1 -> type_of_work == GENERATE)
        comm_num = temp1 -> comm_num;
    else
        if(temp2 -> type_of_work == INSERT)
comm_num = temp2 -> comm_num;
else
  if(temp3 -> type_of_work == FORWARD)
    comm_num = temp3 -> comm_num;

if( transmission_lost_structure[comm_num] == 1 )
  return 1;
else
  return 0;
} //end func
VITA AUCTORIS

Someshwar Roy Choudhury was born in 1974 in Calcutta, India. He graduated from South Point High School in 1992 from Calcutta, India. From there he went on to BMS College of Engg., Bangalore, India, where he obtained a B.E. in Computer Science & Engg. in 1996. He is currently a candidate for the Masters' degree in Computer Science at the University of Windsor and hopes to graduate in Summer 2000.