1996

Torus routing in the presence of multicasts

Hiroki Ishibashi

TORUS ROUTING IN THE PRESENCE OF MULTICASTS

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

By

Hiroki Ishibashi

March 1996

TORUS ROUTING IN THE PRESENCE OF MULTICASTS

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

By

Hiroki Ishibashi

March 1996

Approved by:

[redacted] 3/19/96

Kay Zemoudeh, Chair, Computer Science Date

[redacted]

Arturo I. Concepcion

[redacted]

Owen J. Murphy

ABSTRACT

Three multicast-packet routing algorithms for torus interconnection networks of arbitrary size and dimension are presented. Multicast algorithm 1 uses repeated unicasts to perform multicasts. Multicast algorithm 2 and Multicast Algorithm 3 are new algorithms. These two algorithms are fully adaptive for unicast packets and partially adaptive for multicast packets in the sense that all paths are minimal. Multicast Algorithm 2 requires only three central queues, an injection queue (input buffer), and a delivery queue (output buffer) per node. Multicast Algorithm 3 requires three more central queues and an extra re-injection queue per node. The number of required central queues per node for both Multicast Algorithms 2 and 3 are constant regardless of the size and dimension of the torus network. In the presence of a large number of multicasts on large networks, the third multicast algorithm performs close to the unicast algorithm. Since these algorithms are based on small-sized packet switching method, they are applicable to both multicomputer and Asynchronous Transfer Mode (ATM) switch design. A new technique to build scalable torus networks is also presented.

ACKNOWLEDGMENTS

I would like to thank all faculty members for the excellent education. I would especially like to express my gratitude to Dr. Kay Zemoudeh who patiently helped me in completing this work. Also, I would like to repeat my thanks to Dr. Arturo Concepcion and Dr. Owen Murphy who reviewed this work.

Finally, I gratefully acknowledge the support of my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

a. LIST OF FIGURES

b. LIST OF GRAPHS

CHAPTER 1 -- INTRODUCTION

Parallel computers with binary hypercube interconnection networks have been widely studied in the last decade. Several commercial products, such as the iPSC/860 from Intel Corporation, the nCUBE from nCUBE Corporation, and the CM-2 from Thinking Machine Corporation [19], were developed based on binary hypercube interconnection networks. In massively parallel processing (MPP) computers, interconnection network scalability is important. Binary hypercubes are not scalable. As the number of dimension in the binary hypercube grows, the number of nodes increases exponentially. Parallel computers with mesh and torus interconnection networks are more desirable because of their scalability property [26]. d-dimensional mesh and tori can be laid out in d dimensions using short wires. They can be built using identical boards, each of which requires a small number of pins for connections to other boards [5]. Example machines include the Paragon from Intel Corporation [17], [19], [29], and the T3D and the T3E from Cray Research [19].

The primary disadvantages of mesh and torus interconnection networks are their relatively large diameter and relatively small bisection width [5], [19], [28]. When a network is cut into two equal halves, the minimum number of edges (channels) along the cut is called the bisection width. The diameter of a network is the maximum shortest path between any two nodes. These two network properties of mesh and torus interconnection networks limit the ability of global communications, such as multicasts and broadcasts. However, torus networks have approximately twice the bisection width compared with that of equal sized mesh networks [5], [19]. In addition, the node symmetry of the torus network eliminates congestion from edge nodes of an equal sized and shaped mesh network [5].

1.1 MOTIVATION

A requirement for any routing algorithm is to deliver all messages to the correct destinations without deadlock, livelock, and starvation. The performances of parallel computers mainly depend on the performance of its communication network. Extensive research studies have been completed on torus networks to develop efficient routing algorithms. Most of the existing routing algorithms for mesh and torus networks do not consider multicasts [4], [5], [7], [10], [12], [13], [14], [25], [27]. Multicasts are one-to-many communications. It is still possible to perform multicasts by sending multiple unicasts, but this method increases network latency and causes network congestion quickly. Broadcasts are one-to-all communications. There are several broadcast algorithms for torus networks [22],

[32]; however, they are for wormhole routing. Most broadcast algorithms cannot handle multicasts. Wormhole routing is an efficient technique to hide network latencies for large messages. In wormhole routing, each node has a buffer which is normally less than the size of messages. Also, wormhole routing routes a message in a pipeline fashion. Due to these two properties of wormhole routing, multiple links can be occupied by just one message. This is the primary cause of low utilization of the channels. When the focus is on small-sized packets (for example 57 bytes), the complexity of wormhole routing is wasted. These broadcast algorithms are not efficient and even not applicable to small packet switching. Packet switching is a technique for routing small packets (or messages). It is necessary to develop an efficient multicast algorithm for small packet routing on a torus network. Broadcast is a special case of multicast. Therefore, the multicast algorithm also support broadcasts. The main thrust of this work is to develop an efficient multicast algorithm for small packet switching with minimum network latency. The multicast algorithms presented here are based on the unicast algorithm for packet switching by Cypher and Gravano [5], a fully-minimal-adaptive routing algorithm. Also, a new technique to build scalable folded torus networks is presented. Since ATM cells are small 53 byte messages, the

applicability of the multicast algorithms to ATM switches is also studied. Most ATM switches are based on the multistage interconnection networks (MIN) [1], [8], [20], [24] or fast time multiplexed buses. MINs are dynamic networks [18], [19]. Here the application of static networks, including torus networks, to ATM switches is also studied.

1.2 ORGANIZATION OF CHAPTERS

Chapter 2 introduces different types of switching methods, hypercube interconnection networks, and routing protocols. The definitions of deadlock, livelock, and starvation are given. Also, adaptive routing protocols are described. They are necessary to understand any routing algorithm on a torus network. In Chapter 3, the unicast algorithm by Cypher and Gravano [5] is presented and the fundamental definitions are given. Chapter 4 describes a new technique to build scalable folded torus networks. The simulation method and the performance of the unicast algorithm are studied in this chapter as well. In chapter 5, the formal definitions of Multicast Algorithm 1, Multicast Algorithm 2, and Multicast Algorithm 3 are given. Also, the proof of correctness for Multicast Algorithms 2 and 3 are presented. These algorithms are extensively compared with one another based on the results of simulations. Chapter 6 includes some possible extensions, future work, and conclusions.

CHAPTER 2 -- PRELIMINARIES

In this chapter, fundamental knowledge, which is necessary to discuss any routing protocol on a torus network, is presented.

2.1 SWITCHING METHODS

In this section, several switching methods are described and compared. A switching method is a mechanism to transport information across a network. Network latency is the amount of time required to transport a message from its source to its destination.

In *circuit switching*, a complete path of communication links must be setup between two nodes, the source node and the destination node, prior to the actual communication. This technique is based on the telephone switching method used in most of the existing telephone networks [21]. Once the path between two nodes is set up, there is no need for further signaling or addressing. The minimum network latency of circuit switching is proportional to

$$2N_h \times S_s + L_m$$

where N_h is the number of hops, S_s is the size of signal, and L_m is the length of the message (Figure 2.1). The number of hops is equal to the number of time that a message is transferred between two adjacent nodes. Circuit switching can cause a low channel utilization because once links are in use, no other node can use those links even if they are idle [21]. Since the network latency is dominated by the time required to setup a connection and links are used by only two communicating nodes, this method is advantageous for infrequent long messages. For frequent short messages, there are too many overheads involved to establish a connection beforehand. Therefore, circuit switching is not suitable for small messages (packets).

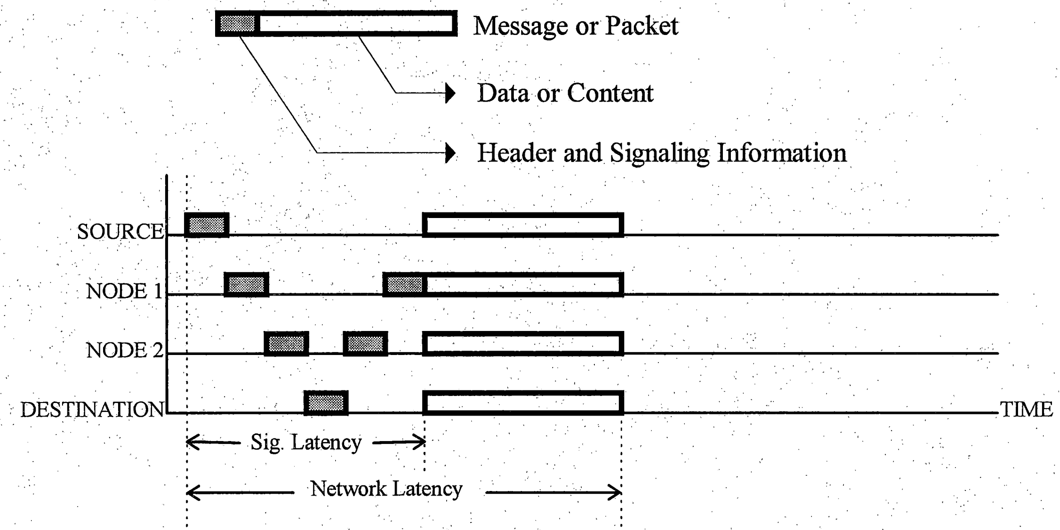


Figure 2.1. Network Latency of Circuit Switching.

In *message switching(Store-and-Forward)*, messages are routed toward their destination nodes without establishing a path. Message switching achieves a better channel utilization than circuit switching by utilizing idle periods of circuit switching [21]. By including addressing information in the header, each message is routed toward its destination dynamically by intermediate nodes. When a message is received in an intermediate node, the message is stored in a buffer temporarily and then is forwarded to a selected adjacent node. The name "store-and-forward" is derived from this routing characteristic. In this method, each link is statistically shared by many nodes. Because each message needs to be received completely at intermediate nodes before it is forwarded to the next node, the communication latency is much higher. The minimum network latency is proportional to

$$N_h \times (S_h + L_m)$$

where S_h is the size of the header (Figure 2.2). In this method, buffers in the nodes must be able to store the longest message allowed.

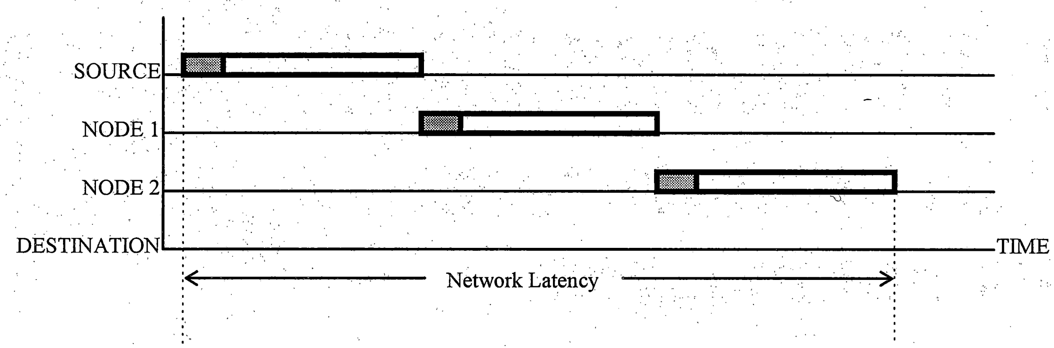


Figure 2.2. Network Latency of Message Switching.

Packet Switching is an improvement over message switching by dividing a message into smaller packets. Each packet has its own addressing information. This introduces additional overhead, but the simultaneous use of links on a path by a message is possible. Packet switching utilizes the communication links more efficiently than message switching [21]. A higher channel utilization and low network latency are possible. The minimum network latency is proportional to

$$N_h \times (S_h + L_p) + (N_p - 1) \times (S_h + L_p) \;=\; (S_h + L_p) \times (N_h + (N_p - 1))$$

where L_p is the length of packet without header (Figure 2.3). The required buffer size is the packet size. Store-and-forward or packet switching is more suitable for ATM traffic since ATM cells (packets) are small. In general, store-and-forward and packet switching are simple techniques which work well when messages or packets are small in comparison with the channel widths [5]. If the messages themselves are small and fixed size, it is possible to apply store-and-forward directly to the algorithms presented here. An example of this scenario is ATM cells.

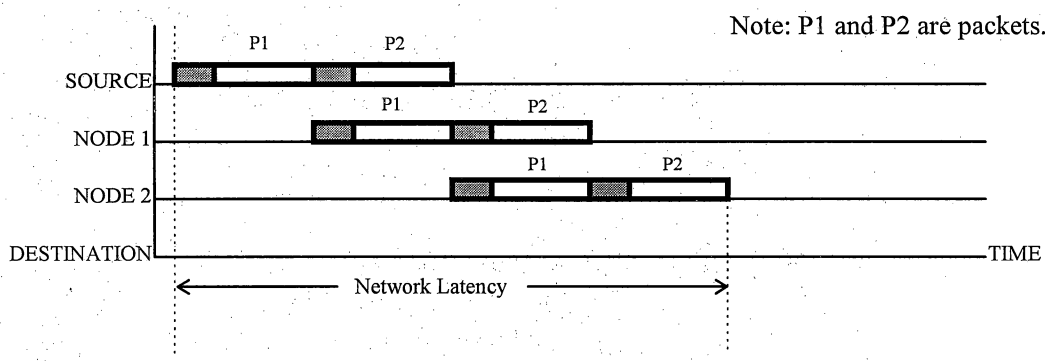


Figure 2.3. Network Latency of Packet Switching.

Virtual cut-through is a mixture of circuit switching and packet switching. Virtual cut-through attempts to overcome the extra latency that is introduced by message switching and packet switching. It permits a message to be transmitted to the next node before it is received completely. The message or packet is divided into smaller units called *flow control units* or *flits* [11], [16], [21]. When enough information for routing is received and the selected outgoing channel is free, the transmission of the flits to the next node starts. Once a message header(flit) is accepted by the next node, the rest of the message or packet follows the same path. Only when the outgoing channel is busy, the message or packet needs to be stored at the blocked intermediate node completely. On a heavily loaded network, virtual cut-through performs similarly to message switching or packet switching. On a lightly loaded network, virtual cut-through performs similarly to circuit switching [21]. The minimum network latency is proportional to

$$N_h \times S_h + L_m$$

refer to Figure 2.4. Virtual cut-through is suitable for lightly loaded networks, and it hides network latency. If the messages or packets are small, there are small differences between store-and-forward or packet switching and virtual cut-through. While packet switching is the simplest and most efficient method for small packets, it is possible to use the algorithms presented here with virtual cut-through.

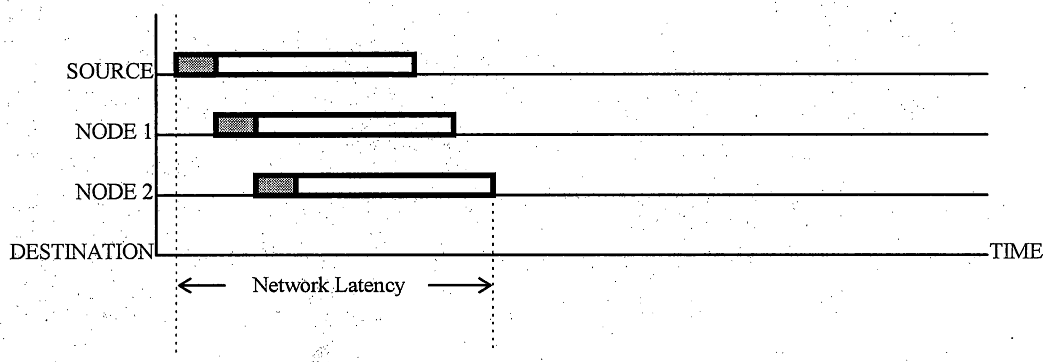


Figure 2.4. Network Latency of Virtual Cut-Through and Wormhole.

Wormhole routing is similar to virtual cut-through with a smaller buffer size. Virtual cut-through requires buffers that are large enough to hold a complete packet or message. Wormhole routing requires buffers that are the size of a message header(flit). Wormhole routing reduces the required size of the buffer in each node; however, there is a drawback to the reduction of the buffer. When an outgoing channel is busy, other channels currently used by the message cannot be freed, unlike virtual cut-through [11]. At light loads, wormhole routing behaves similar to virtual cut-through. Under heavy loads, wormhole routing under-utilizes the networks because of its blocking nature of channels, and it does not perform similarly to message switching or packet switching [11]. The minimum network latency is the same as virtual cut-through. If heavy traffic is expected or traffic is bursty in nature, wormhole routing should not be used. In general, routing algorithms for packet switching and wormhole routing are not interchangeable without modifications.

The hierarchy of switching methods is given in Figure 2.5. The arrows imply inheritances. For example, packet switching inherits its fundamental switching properties from message switching.

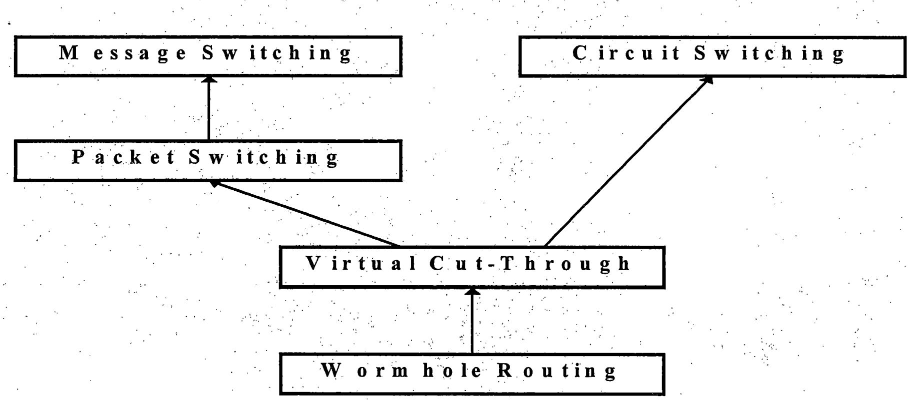


Figure 2.5. Hierarchy of Switching Methods.

2.2 HYPERCUBE INTERCONNECTION NETWORKS

This section formally defines Hypercube Interconnection Networks (HIN). Many interconnection networks, including torus and binary hypercube interconnection networks, belong to the class of HIN.

Let N be the number of processors in an HIN. N can be represented in a mixed radix form as

$$N = k_{d-1} \times k_{d-2} \times k_{d-3} \times \ldots \times k_0 = \prod_{i=0}^{d-1} k_i$$

where k_i is the number of processors in dimension i. Then, each processor between 0 and $N-1$ can be represented as a d-tuple:

$$(a_{d-1}, a_{d-2}, a_{d-3}, \ldots a_0)$$

where $(0 \le a_i \le k_i - 1)$ and d is the number of dimensions in the network. By setting constraints on the values of d and k_i, and the interconnection of processors, different types of HIN results.

Generalized HINs [3], [11]: each processor is interconnected to every other processor whose address differs in exactly one digit (Figure 2.6), or

$\forall i, (a_{d-1}, a_{d-2}, \ldots, a_i, \ldots, a_0)$ is connected to $(a_{d-1}, a_{d-2}, \ldots, a_i', \ldots, a_0)$ if $a_i \ne a_i'$.

A *Hyper-simplified interconnection network is* a generalized HIN such that for all i in generalized HIN, $k_i = k$, or

$\forall i, (a_{d-1}, a_{d-2}, \ldots, a_i, \ldots, a_0)$ is connected to $(a_{d-1}, a_{d-2}, \ldots, a_i', \ldots, a_0)$ if $k_i = k$ and $a_i \neq a_i'$.

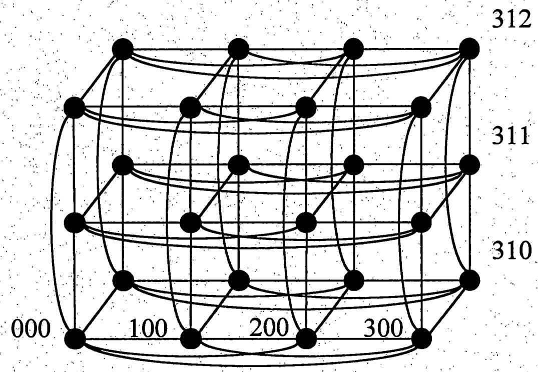


Figure 2.6. Generalized Hypercube.

A *Hyper-rectangular interconnection network* [11] is a generalized HIN where each processor is connected to every other processor whose address differs in exactly one digit by ± 1 modulo the dimension radix (Figure 2.7), or

$\forall i, (a_{d-1}, a_{d-2}, \ldots, a_i, \ldots, a_0)$ is connected to $(a_{d-1}, a_{d-2}, \ldots, a_i', \ldots, a_0)$ if $a_i' = (a_i \pm 1) \bmod k_i$.

In a hyper-rectangular interconnection network, there are cycles in each dimension. An edge between node (0,0,0) and node (3,0,0) is an example of a wraparound connection.

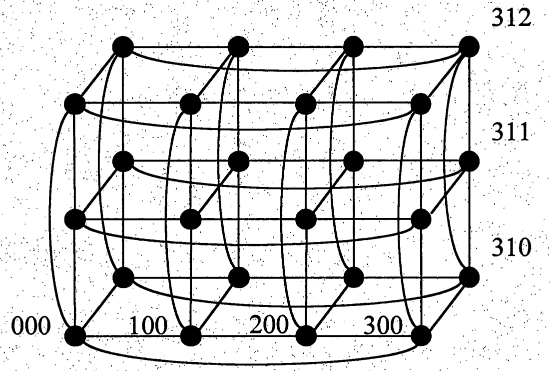


Figure 2.7. Hyper-Rectangular.

k-ary n-cube interconnection networks: for all i, $b_i = k$ and each processor is interconnected to every other processor whose address differs in exactly one digit by ± 1 modulo k, or

$\forall i, (a_{d-1}, a_{d-2}, \ldots, a_i, \ldots, a_0)$ is connected to $(a_{d-1}, a_{d-2}, \ldots, a_i', \ldots, a_0)$

if $k_i = k$ and $a_i' = (a_i \pm 1) \bmod k_i$.

By setting additional constraints on *k*-ary *n*-cube interconnection networks, many well-known interconnection networks can be built. For example, binary hypercubes can be represented by limiting the number of processors in each dimension to two. A 2D torus can be represented by setting the number of dimension to two. Likewise, a 3D torus is represented by setting the number of dimension to three. In general, A hyper-rectangular is called torus. Figure 2.8 shows the taxonomy of HIN.

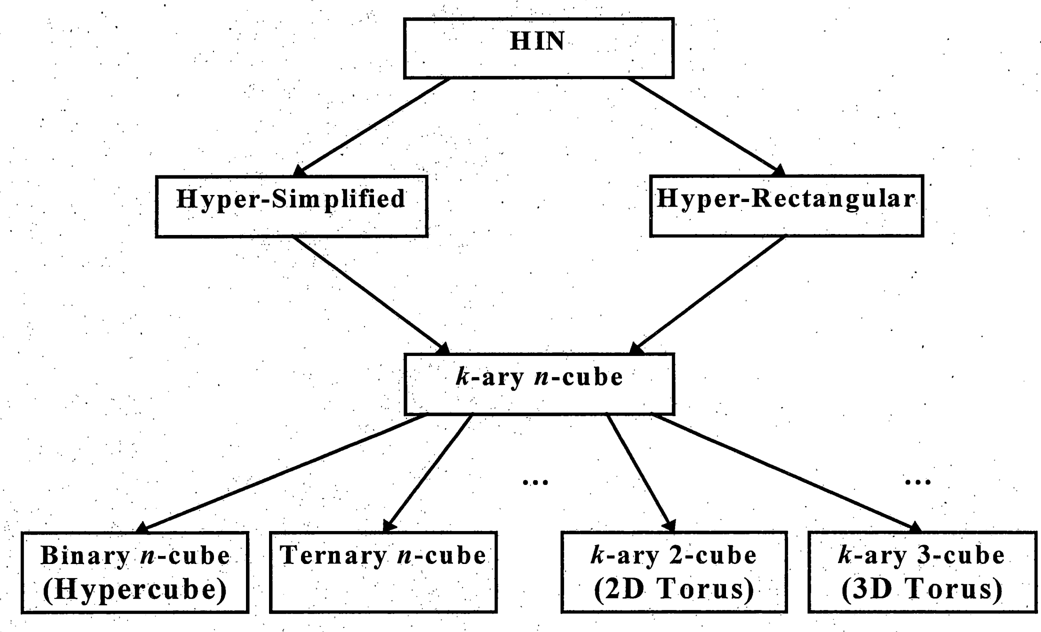


Figure 2.8. Taxonomy of HIN.

2.3 DEADLOCK, LIVELOCK, AND STARVATION

A routing algorithm has to guarantee freedom from deadlock, livelock, and starvation. By avoiding these conditions, a routing algorithm will eventually deliver a message to its destination. The descriptions of deadlock, livelock, and starvation are given below.

Deadlock may occur when the routing protocol waits for the required resources, such as links and buffer spaces, to become available. Deadlock is a situation where no message can move toward its destination because of formation of cyclic dependencies among network resources.

Livelock occurs when a message circulates in a network, never reaching its destination. If a routing protocol does not guarantee minimal paths, then there exists the possibility of livelock.

Starvation occurs when a message waits for its required resources indefinitely while those resources are allocated to other messages.

2.4 ADAPTIVE ROUTING PROTOCOLS

A routing protocol is a set of rules which defines how a message is sent from its source to its destination. Adaptive protocols have the ability to dynamically select possible routes at each intermediate node. A message that is routed by non-adaptive routing protocols can only take a predetermined path. On a large-scale multicomputer, multiprocessor, or network of computers, it is desirable to apply an adaptive routing protocol to make more efficient use of interconnection bandwidth [11]. Adaptive routing protocols are classified as progressive or backtracking. Progressive protocols always try to move forward and have a limited ability to backtrack. Backtracking protocols systematically search the network to find possible paths by backtracking as needed. Backtracking protocols should not be used in networks which require fast routing decisions, but are suited for faulty networks.

Progressive and backtracking protocols are classified as misrouting or profitable. A link, which brings a message closer to its destination, is called a profitable link. A profitable protocol only uses profitable links for routing at each node. A misrouting protocol can use both profitable and non-profitable links. Misrouting might lead to livelock.

Profitable and misrouting protocols are classified as fully or partially adaptive. A fully adaptive protocol can use all paths that are available for routing. A partially adaptive protocol is restricted to use a subset of all paths that are available for routing. If a routing protocol is fully adaptive, profitable, and progressive, it is said to be fully-minimal-adaptive. Figure 2.9 shows the taxonomy of adaptive routing protocols [11].

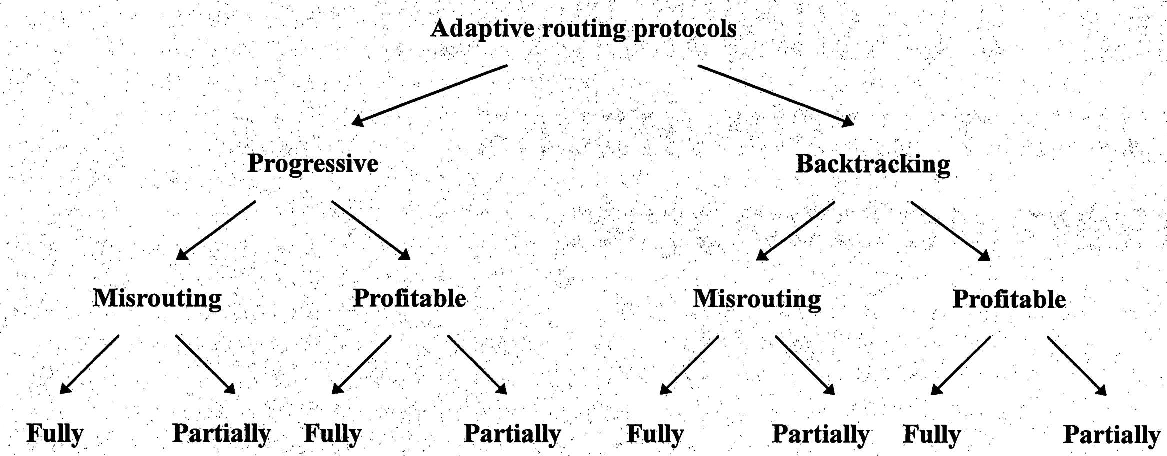


Figure 2.9. Taxonomy of Adaptive Routing Protocols.

CHAPTER 3 -- FULLY-MINIMAL-ADAPTIVE UNICAST ON TORUS NETWORKS

In this chapter, several definitions and assumptions are given. They are necessary to describe the unicast algorithm [5] and the multicast algorithms in Chapter 5. Simulation result of the unicast algorithm is presented and discussed at the end of Chapter 4.

3.1 DEFINITION OF TERMS

Each node in the torus network contains an injection queue, a delivery queue, and three central queues (Figure 3.1). Packets can enter the torus network only by being placed in an empty injection queue in their source node. Also, packets can be removed from the network only at their destination node's delivery queue. The injection queue and delivery queue are introduced to simplify the description of the model. It is not necessary for these two queues to be present. Consequently, only central queues are counted as the number of queues required by a routing algorithm. Each central queue in a node should be directly accessed from all of the node's input ports.

Given the source and the destination node of a packet and the queue in which the packet is currently stored, an adaptive routing algorithm specifies a set of queues to which the packet may be moved next. This set of queues is called the packet's *waiting set*. A waiting set can consist of queues either in the node that currently holds the packet or in neighboring nodes. Injection queues are not allowed to appear in waiting sets. The waiting set of a packet which is currently in a delivery queue must be empty. Injection queues are used only for introducing new packets to the network. Delivery queues are used only for removing packets which have reached their destination. When a packet is moved from one queue to another, it occupies both of the queues for a finite amount of time.

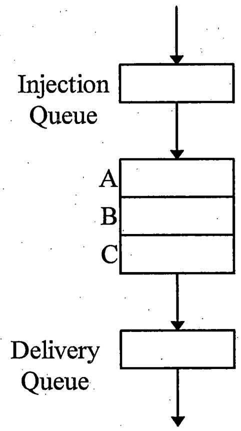


Figure 3.1. The Queue Structure of the Unicast Algorithm.

3.2 ASSUMPTIONS

Several assumptions are made on the torus network properties based on the "well-behaved buffer management" by Günther[15].

A1. No "starvation in poverty." No packet remains in a queue forever while an infinite number of packets enter and leave some queue in its waiting set.

A2. A packet that is in the delivery queue of its destination node will be removed from the network within a finite time.

A3. No "starvation in wealth." No packet remains in a queue forever if there is a queue in its waiting set which is empty or permanently empty.

A1 and A2 ensure that packets never wait for a queue for an infinitely long time without any reason. A3 prevents starvation. Under the assumption of well-behaved buffer management, Günther has proved that a torus routing algorithm is deadlock and starvation free [15].

Lemma 3.1 (Günther): Given a total ordering of the queues in the network, a routing network is free of deadlock and no packet will remain in a single queue under the assumption of well-behaved buffer management if one of the following is satisfied for every packet:

- A packet is in the delivery queue of its destination node.
- A packet has a waiting set that contains a higher ordered queue than the one it occupies currently.

Lemma 3.1 does not force packets to be routed through queues in increasing order; it ensures that every packet always has a chance to move to a higher ordered queue.

3.3 NODE ORDERINGS

Several useful node orderings were introduced by Cypher and Gravano [5]. These node orderings are used to define the queue orderings used in the algorithms introduced here. To describe several node orderings, an 8×9 torus is used as an example (Figure 3.2).

The following four node orderings are defined.

- *Right-increasing* ordering is the simple row-major ordering of the nodes.
- *Left-increasing* ordering is simply the reverse of the right-increasing ordering.
- *Inside-increasing* ordering assigns the smallest values to the nodes near the wraparound edges of the torus and the largest values to the nodes near the center of the torus network.
- *Outside-increasing* ordering is simply the reverse of the inside-increasing ordering.

Refer to Table 3.1, Table 3.2, Table 3.3, and Table 3.4.

Formally given an integer $i, 0 \le i < d$, let

$$g(i) = \prod_{j=0}^{i-1} k_j \quad (g(0) = 1) .$$

For any torus node label of the form $(a_{d-1}, a_{d-2}, \ldots, a_0)$, define

$$Eval((a_{d-1}, a_{d-2}, \ldots, a_o)) = \sum_{i=0}^{d-1} g(i) a_i \ .$$

Function *Eval* assigns a unique integer in the range of 0 through $n-1$ to each node. It interprets a node label as a mixed radix representation of integers. To obtain the four total orderings, the nodes are first re-labeled according to the following functions [5]. Then, the Function *Eval* is used to evaluate the new labels as integers. Given any integer $k_i \geq 2$ and a_i where $0 \leq a_i < k_i$, we have

- $f_R(a_i, k_i) = a_i$ (orders the numbers 0 through $k_i - 1$ in increasing order from left to right),
- $f_L(a_i, k_i) = k_i - a_i - 1$ (orders the number 0 through $k_i - 1$ in increasing order from right to left),
- $f_I(a_i, k_i) = \begin{cases} a_i & \text{if } a_i < \lfloor k_i / 2 \rfloor \\ \lfloor 3k_i / 2 \rfloor - a_i - 1 & \text{otherwise} \end{cases}$ (orders the numbers 0 through $k_i - 1$ from the outside to the inside),
- $f_O(a_i, k_i) = \begin{cases} k_i - a_i - 1 & \text{if } a_i < \lfloor k_i / 2 \rfloor \\ a_i - \lfloor k_i / 2 \rfloor & \text{otherwise} \end{cases}$ (orders the numbers 0 through $k_i - 1$ from the inside to the outside).

Examples of these four functions are given in Table 3.5.

The four functions to produce the total orderings from the mixed radix representation of node labels are defined by

- $$\begin{aligned}&\mathit{Right}((a_{d-1}, a_{d-2}, \ldots, a_0)) \\ &\quad = \mathit{Eval}((f_R(a_{d-1}, k_{d-1}), (f_R(a_{d-2}, k_{d-2}), \ldots, (f_R(a_0, k_0))),\end{aligned}$$
- $$\begin{aligned}&\mathit{Left}((a_{d-1}, a_{d-2}, \ldots, a_0)) \\ &\quad = \mathit{Eval}((f_L(a_{d-1}, k_{d-1}), (f_L(a_{d-2}, k_{d-2}), \ldots, (f_L(a_0, k_0))),\end{aligned}$$
- $$\begin{aligned}&\mathit{Inside}((a_{d-1}, a_{d-2}, \ldots, a_0)) \\ &\quad = \mathit{Eval}((f_I(a_{d-1}, k_{d-1}), (f_I(a_{d-2}, k_{d-2}), \ldots, (f_I(a_0, k_0))),\end{aligned}$$
- $$\begin{aligned}&\mathit{Outside}((a_{d-1}, a_{d-2}, \ldots, a_0)) \\ &\quad = \mathit{Eval}((f_O(a_{d-1}, k_{d-1}), (f_O(a_{d-2}, k_{d-2}), \ldots, (f_O(a_0, k_0))).\end{aligned}$$

A transfer of a packet from node x to an adjacent node y is said to occur to the right if and only if x is smaller than y in the right-increasing ordered torus network (Figure 3.3). Similarly, a transfer of a packet from node x to an adjacent node y is said to occur to the inside if and only if x is smaller than y in the inside-increasing ordered torus network (Figure 3.5). For other orderings, refer to Figure 3.4 and Figure 3.6.

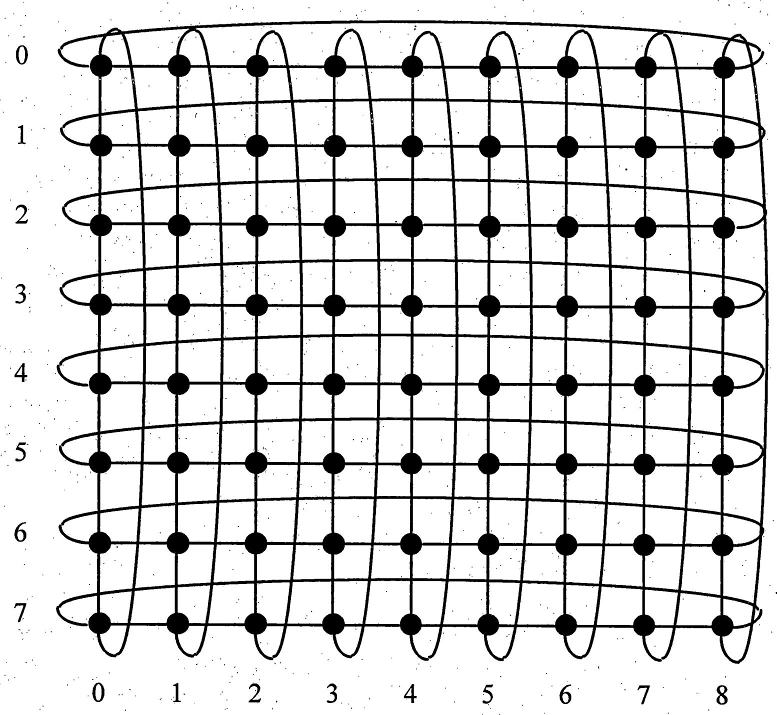


Figure 3.2. 8×9 Torus Network.

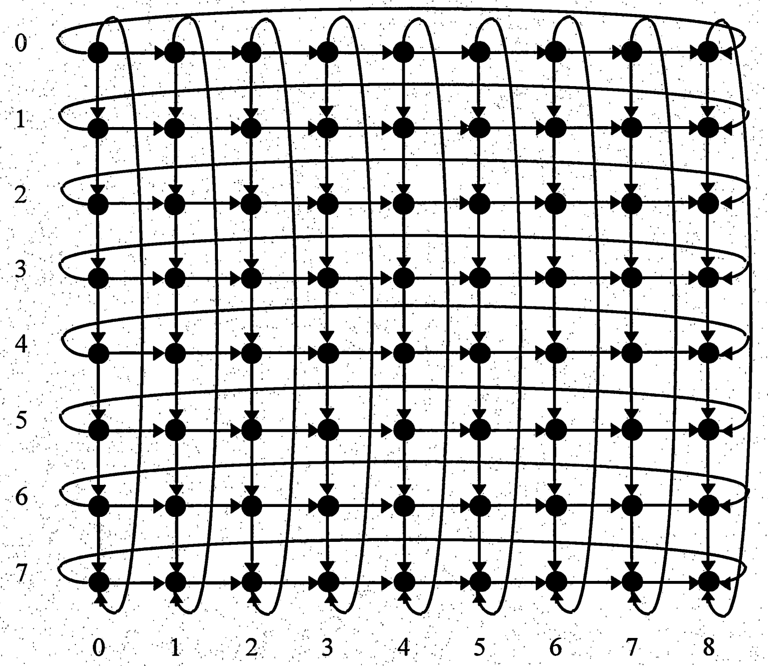


Figure 3.3. 8×9 Torus with Right-increasing Direction Edges.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71

Table 3.1. Right-increasing Ordering in 8×9 Torus.

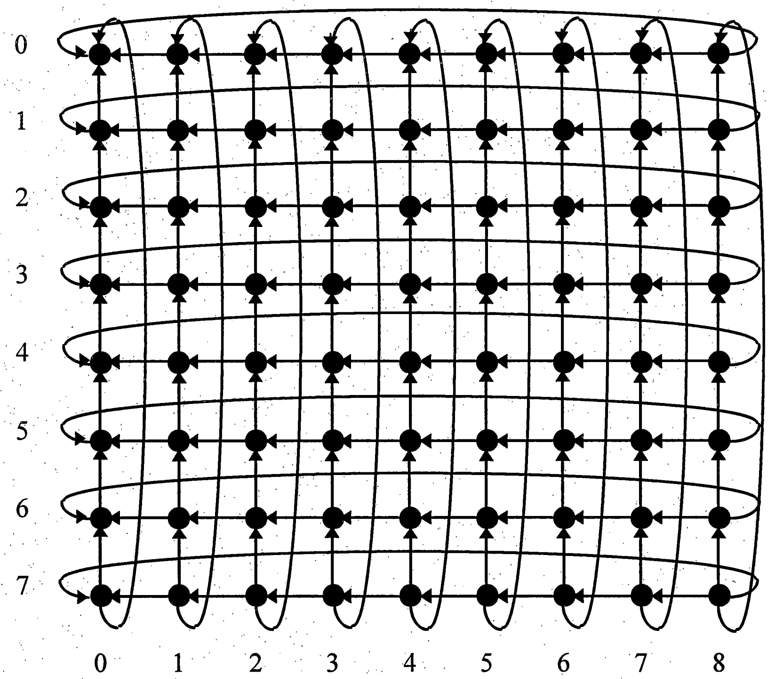


Figure 3.4. 8×9 Torus with Left-increasing Direction Edges.

71	70	69	68	67	66	65	64	63
62	61	60	59	58	57	56	55	54
53	52	51	50	49	48	47	46	45
44	43	42	41	40	39	38	37	36
35	34	33	32	31	30	29	28	27
26	25	24	23	22	21	20	19	18
17	16	15	14	13	12	11	10	9
8	7	6	5	4	3	2	1	0

Table 3.2. Left-increasing Ordering in 8×9 Torus.

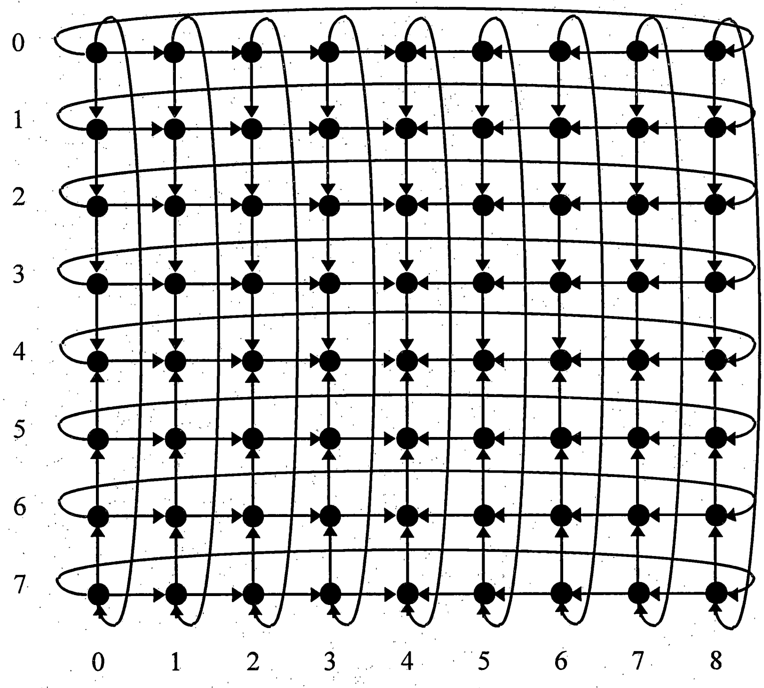


Figure 3.5. 8×9 Torus with Inside-increasing Direction Edges.

0	1	2	3	8	7	6	5	4
9	10	11	12	17	16	15	14	13
18	19	20	21	26	25	24	23	22
27	28	29	30	35	34	33	32	31
63	64	65	66	71	70	69	68	67
54	55	56	57	62	61	60	59	58
45	46	47	48	53	52	51	50	49
36	37	38	39	44	43	42	41	40

Table 3.3. Inside-increasing Ordering in 8×9 Torus.

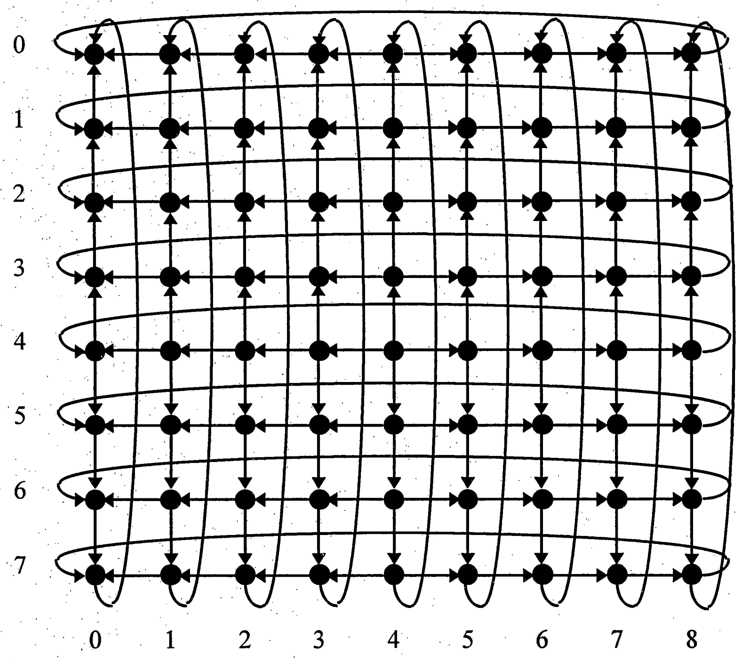


Figure 3.6. 8×9 Torus with Outside-increasing Direction Edges.

71	70	69	68	63	64	65	66	67
62	61	60	59	54	55	56	57	58
53	52	51	50	45	46	47	48	49
44	43	42	41	36	37	38	39	40
8	7	6	5	0	1	2	3	4
17	16	15	14	9	10	11	12	13
26	25	24	23	18	19	20	21	22
35	34	33	32	27	28	29	30	31

Table 3.4. Outside-increasing Ordering in 8×9 Torus.

a_i :	0	1	2	3	4	5	6	7	8
$f_R(a_i, 9)$:	0	1	2	3	4	5	6	7	8
$f_L(a_i, 9)$:	8	7	6	5	4	3	2	1	0
$f_I(a_i, 9)$:	0	1	2	3	8	7	6	5	4
$f_O(a_i, 9)$:	8	7	6	5	0	1	2	3	4

Table 3.5. The Functions $f_R(a_i,k_i)$, $f_L(a_i,k_i)$, $f_I(a_i,k_i)$, and $f_O(a_i,k_i)$ When $k_i = 9$.

3.4 NOTATION

The following notations are used in the algorithms described here. Let *p* be an arbitrary packet that is being routed in a torus network.

queue(p)	The queue in which *p* is currently stored.
node(p)	The node in which *p* is currently located.
source(p)	*p*'s source node.
destination(p)	*p*'s destination.
wait(p)	*p*'s waiting set. A waiting set consists of the set of queues to which the packet may be moved next.
neighbors(p)	The set of nodes that are torus neighbors of *node(p)*.
ok_nodes(p)	Subset of *neighbors(p)* consisting of those neighboring nodes that lie along a minimal length path from *node(p)* to *destination(p)*.
ok_queues(p)	The set of central queues in *ok_nodes(p)* that are directly accessible from *node(p)*.

3.5 THE UNICAST ALGORITHM

A minimal-fully-adaptive packet routing algorithm for unicasts is introduced by Cypher and Gravano [5]. This algorithm is proved to be deadlock, livelock, and starvation free based on the well-behaved buffer management assumption [15]. The advantage of this algorithm is that it requires only three central queues per node regardless of the size and dimension of the torus network. For example, "hop-so-far" scheme [25] requires larger queues than the diameter of the torus. The Ngai and Dhar algorithm [27] is a novel approach to avoid deadlock by tokens, but it requires more buffers to route packets efficiently as the diameter increases.

The fully-minimal-adaptive algorithm for unicastings by Cypher and Gravano is presented here. The algorithm is run on every node to find *wait(p)* of any packet *p*. *ok_nodes(p)* is calculated on each node using *destination(p)* and *node(p)* every time *p* moves between any two queues as follows.

> Let *destination(p)* be $(a_{d-1},\ldots,a_i,\ldots,a_0)$ and *node(p)* be $(b_{d-1},\ldots,b_i,\ldots,b_0)$. Let length = $a_i - b_i$.
>
> For i = 0 to $d-1$, do the following to find nodes to be included in *ok_nodes(p)*.
>
> If k_i modulo 2 = 0 AND |length| = $\lfloor k_i/2 \rfloor$ Then
>   Include both positive and negative adjacent nodes on dimension i.
> Else if length > 0 Then
>   If length ≤ $\lfloor k_i/2 \rfloor$ Then

```
        Include an adjacent node in the positive
        direction on dimension i.
      Else
        Include an adjacent node in the negative
        direction on dimension i.
      End if
   Else if length < 0
      If |length| ≤ ⌊k_i/2⌋ Then
        Include an adjacent node in the negative
        direction on dimension i.
      Else
        Include an adjacent node in the positive
        direction on dimension i.
      End if
   End if
```

For example, on Figure 3.2, let *node(p)* be node (5,5) and *destination(p)* be node (2,3). In this case, nodes (5,4) and (4,5) are in *ok_nodes(p)*. Based on *wait(p)* and the current condition of a network, *node(p)* decides the next movement of packet *p* dynamically.

Unicast Algorithm:
Let A, B, and C be three central queues required by the algorithm (Figure 3.1). Let *p* be an arbitrary packet that is being routed by the algorithm. Let *q* = *queue(p)*, and *x* = *node(p)*. The algorithm creates *p*'s waiting set *(wait(p))* according to the following rules.

Case 1: *q* is an injection queue.
In this case, *wait(p)* consists of the A queue in x.
Case 2: *q* is an *A* queue.
In this case, there are two subcases.

Case 2a: $\exists y \in ok\_nodes(p)$ **such that** $Right(x) < Right(y)$.
In this subcase, *wait(p)* consists of all of the *A* queues in *ok_queues(p)*.

Case 2b: $\nexists y \in ok\_nodes(p)$ **such that** $Right(x) < Right(y)$.
In this subcase, *wait(p)* consists of the *B* queue in x.

Case 3: *q* is a *B* queue.
In this case, there are two subcases.

Case 3a: $\exists y \in ok\_nodes(p)$ **such that** $Left(x) < Left(y)$.
In this subcase, *wait(p)* consists of all of the *B*

queues in *ok_queues(p)*.

Case 3b: $\nexists y \in ok\_nodes(p)$ **such that** $Left(x) < Left(y)$**.**
In this subcase, *wait(p)* consists of the *C* queue in x.

Case 4: *q* **is a** *C* **queue.**
In this case, there are two subcases.

Case 4a: $x \neq destination(p)$**.**
In this subcase, *wait(p)* consists of all of the *C* queues in *ok_queues(p)*.

Case 4b: $x = destination(p)$**.**
In this subcase, *wait(p)* consists of the delivery queue in x.

Case 5: *q* **is a delivery queue.**

Consider a packet *p* that is routed from source node (4,7) to destination node (2,2) in an 8×9 torus network. Figure 3.7 represents one possible minimal path. The sequence of packet movements in the queues is the injection queue of node(4,7) to the A queue of node(4,7) to the A queue of node(3,7) to the A queue of node(3,8) to the B queue of node(3,8) to the B queue of node(3,0) to the B queue of node(2,0) to the C queue of node(2,0) to the C queue of node(2,1) to the C queue of node(2,2) to the delivery queue of node(2,2). The correctness of the algorithm is proven [5].

The queue structure in each node should accommodate multiple injection and delivery queues to prevent loss of incoming and outgoing packets as in Figure 3.8. There is no need to change the algorithm to handle multiple injection and delivery queues.

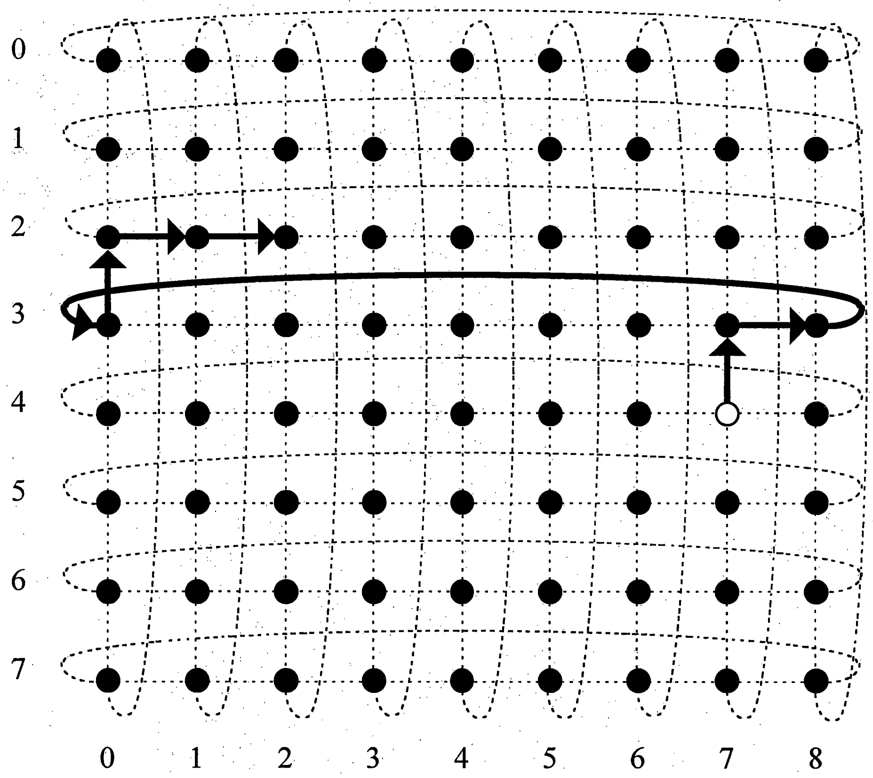


Figure 3.7. An Example of a Route by the Unicast Algorithm.

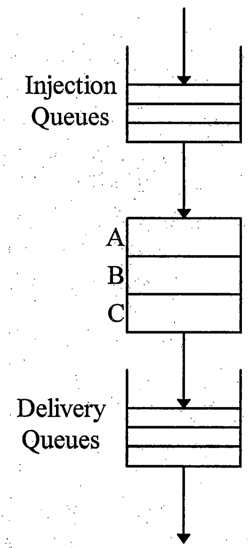


Figure 3.8. The Modified Queue Structure.

CHAPTER 4 -- IMPLEMENTATION AND PERFORMANCE EVALUATION OF THE UNICAST ALGORITHM

In this chapter, a technique to build scalable folded torus networks is presented. Base units to build 1-, 2-, and 3-dimensional folded torus networks and the architectures of the base units used in the algorithms are described. The performance characteristics of the unicast algorithm are also presented.

4.1 NODE ARCHITECTURE AND ORGANIZATION

The simplicity of the interconnections between nodes is the primary advantage of the torus network [2], [5], [19]. On building large-scale parallel computers, the complexity of wiring between nodes becomes an important issue. The cost of *k*-ary *n*-cube networks is dominated by the amount of wire, rather than the number of switches required [19]. An efficient method of building torus networks is prerequisite.

4.1.1 FOLDED TORUS NETWORKS

Consider a linear array interconnection network as in Figure 4.1. By adding a wraparound connection between node 0 and node n-1 in an n node linear array, a 1-dimensional torus (1-D torus or ring) can be realized (Figure 4.2). Note that the wraparound link is longer than other links. This results in longer communication latency along the wraparound edge. To equalize the length of all links, the torus is folded along its bisection link of its underlying linear array, resulting in a perfect shuffle of nodes as in Figure 4.3. This is the 1-D folded torus network.

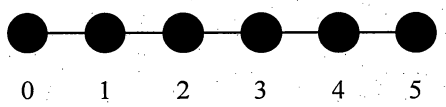


Figure 4.1. Linear Array.

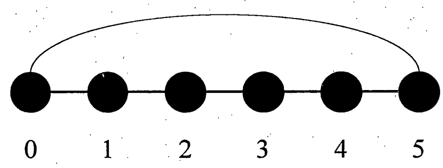


Figure 4.2. 1-D Torus.

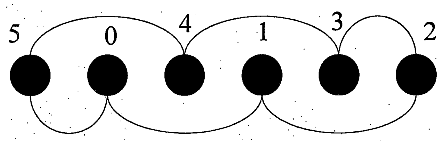


Figure 4.3. 1-D Folded Torus.

4.1.2 BASE UNITS

A 1-D folded torus network can be built out of scalable base units. Figure 4.4 indicates the base unit for 1-D folded torus networks. For example, a 6-node 1-D torus can be built using three base units, and two end pins that are placed at both ends (Figure 4.5). Such networks are easily modifiable. To add more nodes to an existing network, additional base units are inserted between an end pin and the base unit next to the end pin. Possible network sizes are multiples of two nodes.

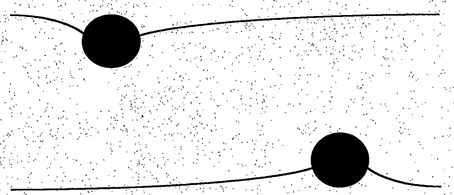


Figure 4.4. Base Unit for 1-D Torus.

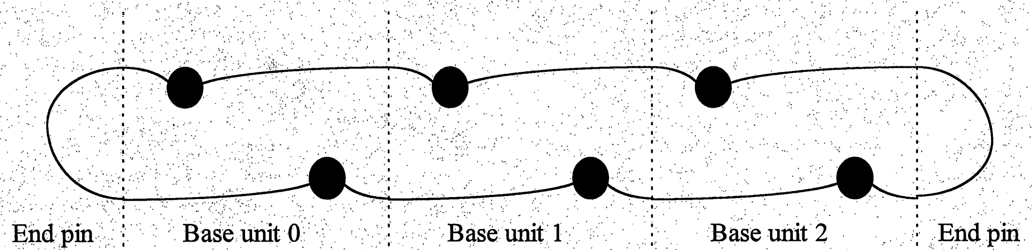


Figure 4.5. 3 Base Units and 2 End Pins.

Similarly, a 2-D folded torus can be built using the base unit with four nodes (Figure 4.6).

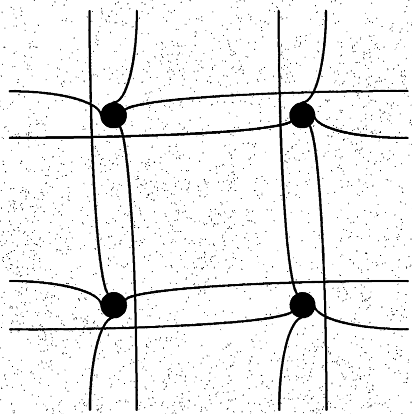


Figure 4.6. Base Unit for 2-D Torus.

A 6×6 folded network can be built using 9 base units (Figure 4.7). At a glance, folded torus networks seem to be different from torus networks that were introduced in Chapter 3. However, they are topologically equivalent [19] and any algorithm that runs on a torus network also runs, without any modification, on an equivalent folded torus network. For the 2-D torus, possible network sizes are $2m \times 2n$, $m \geq 1$ and $n \geq 1$. m and n should be kept as close to each other as possible to avoid large diameters. For example, the sequence of 2×2, 2×4, 2×6, 4×4, 4×6, 4×8, 6×6, etc. is the desirable way to scale up the 2-D torus network.

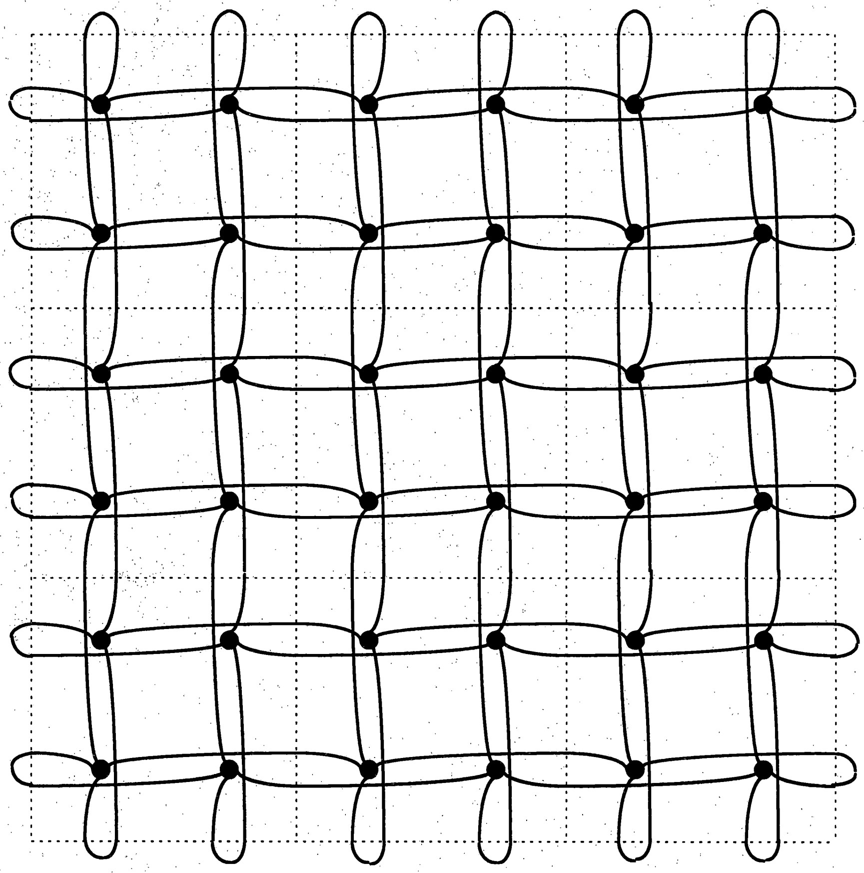


Figure 4.7. 2×2 Base Units in a 6×6 Folded Torus.

For the 3-D torus, a base unit consists of eight nodes (2×2×2) as in Figure 4.8. Figure 4.9 is an example of a 2×4×4 torus network using four such base units. Similar to 2-D base units, it is desirable to keep the diameter of the networks small as possible. Possible network sizes are $2n \times 2m \times 2l$ where $n \geq 1$, $m \geq 1$, and $l \geq 1$. For example, the sequence, 2×2×2, 2×2×4, 2×2×6, 2×4×4, 2×4×6, 4×4×4, etc., is the desirable way to scale up the 3-D torus network.

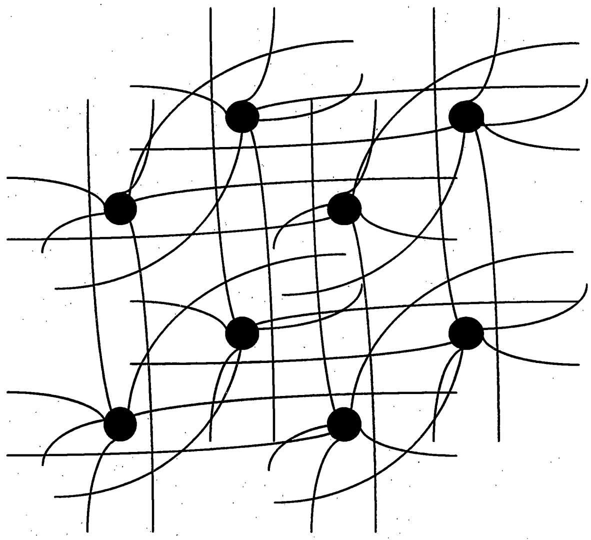


Figure 4.8. Base Unit for 3-D Torus.

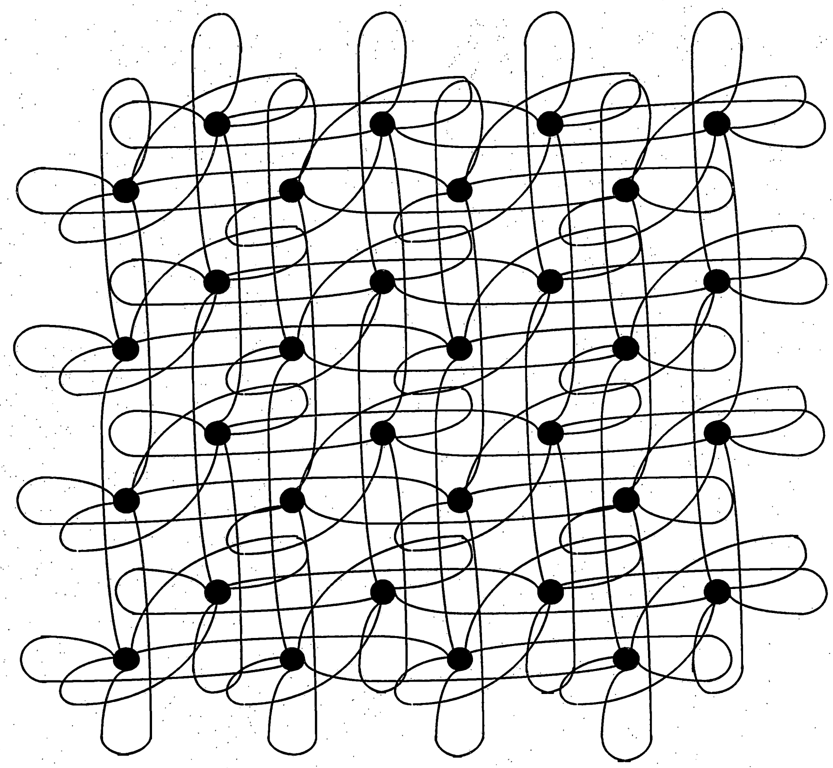


Figure 4.9. 2×2×2 Base Units in a 2×4×4 Folded Torus.

4.1.3 ARCHITECTURE OF BASE UNITS

The symmetry of folded torus networks makes them ideal for VLSI implementation. Figure 4.10 shows the implementation of a single node. Within a chip, queues are hard-wired. Each queue has a tag (T). Tags are used to indicate whether a queue is occupied or not. By checking the tag of the next node's queue, neighboring nodes can directly send a packet to the next node. Injection and delivery queues are implemented as expandable caches, either on or off the chip. Each chip is self-clocked, otherwise, it would be difficult to synchronize all nodes on large networks [6], [9]. To implement the 1-D base unit, two of these nodes are placed in one chip. For the 2-D base unit, four of these nodes are placed in one chip. Similarly, eight of these nodes are placed in one chip for the 3-D base unit.

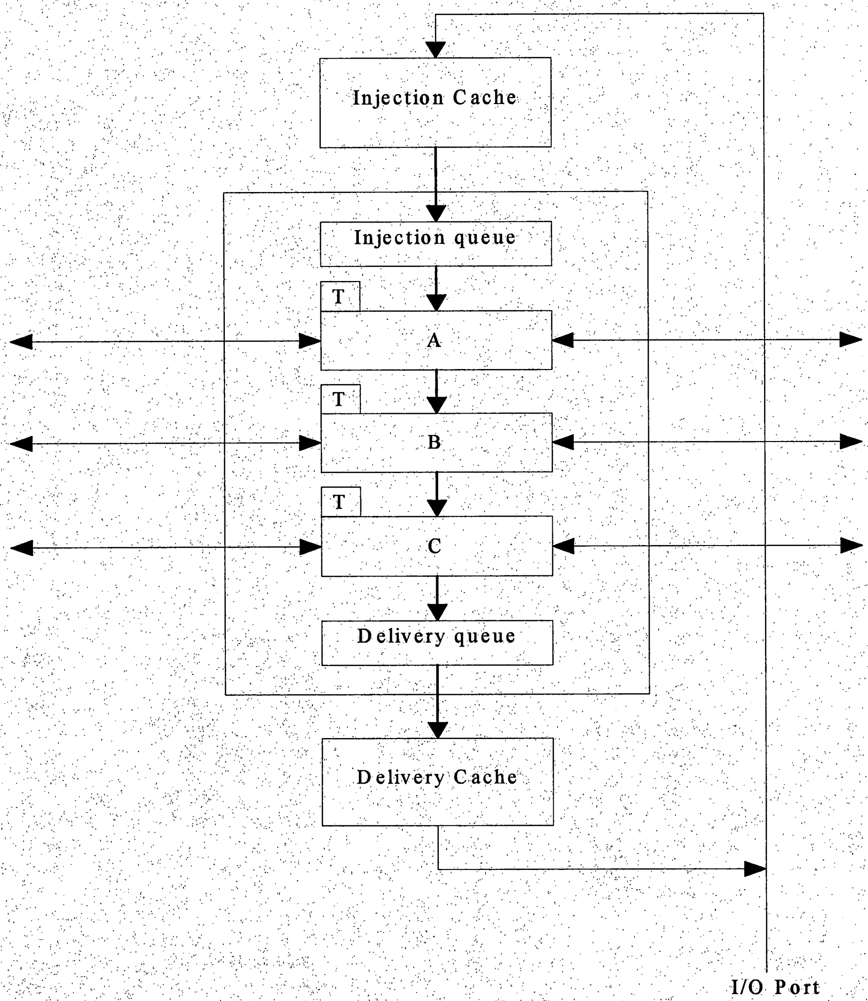


Figure 4.10. Single Node Implementation on a Chip.

4.2 SIMULATION METHOD

In this section, several important properties of the simulation are discussed. The simulation method, which is introduced in this section, is used for both the unicast algorithm and the multicast algorithms.

4.2.1 PREVENTING STARVATION

On simulating the unicast algorithm, the assumptions of well-behaved buffer management that were made in Section 3.2 need to be implemented. To prevent starvation, priorities are assigned to each incoming link of A, B, and C queues. The priorities are examined in a round robin fashion. For example, each A queue on a 2-D torus network has an incoming link from its injection queue and the A queues of its north, east, south, and west neighbors. An example of the priorities of the A queue is shown in Figure 4.11.

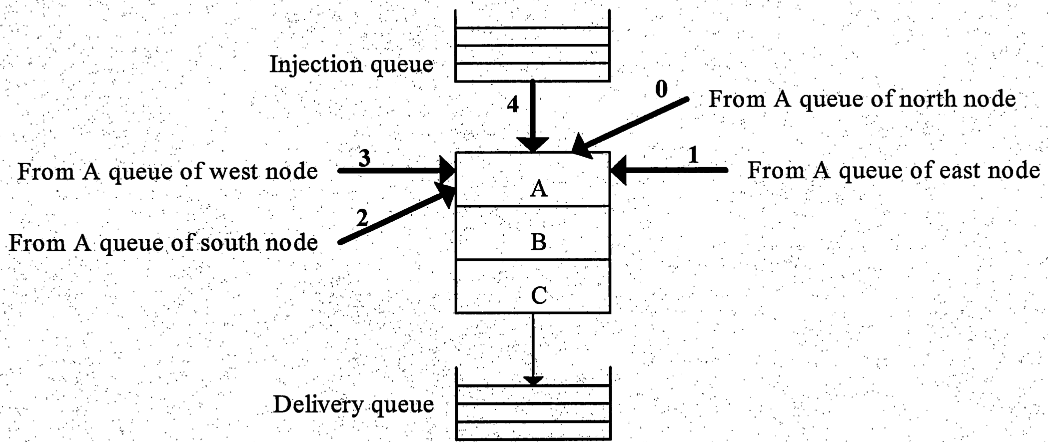


Figure 4.11. An Example of Packet Priorities.

With these priorities, if there are packets from the east and the north neighbors, the packet from the east neighbor will be placed in the A queue, since it has a higher priority. After the packet is placed in the A queue, the priorities are rotated in a clockwise fashion. This priority scheme ensures the fairness. Similarly, the B and C queues use the same priority scheme.

4.2.2 SIMULATION PROPERTIES

Packets will arrive at the injection queues based on the negative exponential distribution with mean inter-arrival time = $1/\lambda$. Each node has its own λ. Packets are removed from the delivery queues based on the negative exponential distribution with mean = $1/\mu$. Each node has its own μ. The size of each packet is 57 bytes. This packet size is based on the size of ATM cells (53 bytes). It includes 4 more bytes in the header to include routing information. The inter-queue latency is the amount of time required to move a packet between two queues on the same node. 100 ns is assigned to the inter-queue latency. The inter-node latency, which is the amount of time required to move a packet between two nodes, is 450 ns on average. This average inter-node latency is calculated based on the architecture of the base unit. To move a packet between base units, an 800 ns latency is assumed. Within a base unit, the inter-node latency is 100 ns. The probability of sending a packet to a node outside of a base unit is 0.5. Similarly, the probability of sending a packet within a base unit is 0.5. Therefore, the average inter-node latency is obtained by

$$100\,ns \times 0.5 + 800\,ns \times 0.5 = 450\,ns\ .$$

Consequently, the channel bandwidth is calculated as 1 Gbps by

$$\frac{57\ bytes \times 8\ bits}{450\ ns} \approx 1\ Gbps\ .$$

Network latency is measured from the moment when a packet is placed in the injection queue until its arrival at the delivery queue.

Network throughput is calculated as

$$Network\ throughput = \frac{number\ of\ packets\ delivered}{unit\ time}\ .$$

Queue utilization is the percent of the time when central queues are occupied. Since each node only manipulates its central queues, the queue utilization is a good indication of the node utilization.

4.2.3 SIMULATION PATTERNS

Three simulation patterns are prepared for a 4×4×4 torus network and a 8×8×8 torus network. Pattern 1 creates moderate traffic. Pattern 2 creates medium traffic. Pattern 3 creates heavy traffic. A set of λ and μ is assigned to each simulation. Based on λ and μ, new rates, λ' and μ', are assigned to each node as follows.

- *Pattern 1 (Moderate Traffic):*

  4x4x4 torus network:

1 node	$\lambda' = \lambda$
2 nodes	$\lambda' = \lambda/10$
61 nodes	$\lambda' = \lambda/100$

  8x8x8 torus network:

1 node	$\lambda' = \lambda$
11 nodes	$\lambda' = \lambda/10$
500 nodes	$\lambda' = \lambda/100$

  $\mu' = \mu$ for all nodes on both torus networks.

- *Pattern 2 (Medium traffic):*

  For both 4x4x4 and 8x8x8 torus networks:

¼ nodes	$\lambda' = \lambda$
¼ nodes	$\lambda' = \lambda/2$
¼ nodes	$\lambda' = \lambda/4$
¼ nodes	$\lambda' = 3\lambda/4$

  $\mu' = \mu$ for all nodes.

- *Pattern 3: (Heavy traffic):*

For both 4x4x4 and 8x8x8 torus networks:

Each node is randomly assigned λ' based on the negative exponential distribution with mean = λ.

Each node is randomly assigned μ' based on the negative exponential distribution with mean = μ.

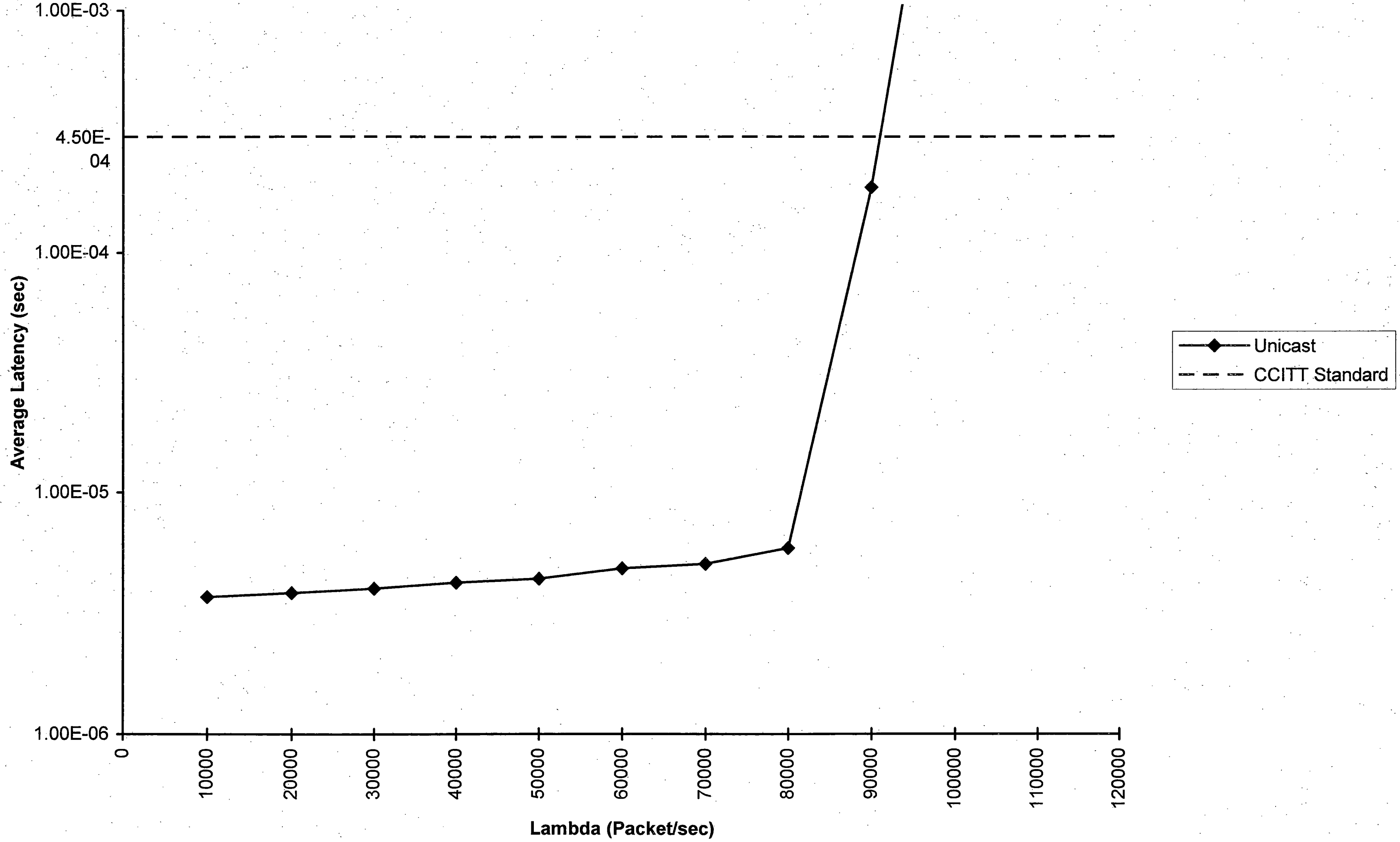
It is important to note that λs on the graphs in the following sections and chapters do not indicate the average λ' for each pattern. The average λ' (the actual input rate) is calculated by taking the average of λ' of all nodes. For example, the calculation of the average λ' of pattern 2 for a 4×4×4 torus network is

$$Average\ \lambda' = \frac{16\times\lambda+16\times\lambda/2+16\times\lambda/4+16\times3\times\lambda/4}{64}.$$

4.3 SIMULATION RESULT OF THE UNICAST ALGORITHM

Graph 4.1 is the result of the simulation for the unicast algorithm using pattern 3 (heavy traffic) on the 8×8×8 torus network. The Consultative Committee on International Telegraphy and Telephony (CCITT) defines the average allowable latency of 450 μs for ATM switches [8]. This limit is indicated on all the graphs presented here. Any latency beyond this limit is unacceptable. The result of the unicast simulation is used to compare the latency of the multicast algorithms in the next chapter.

Graph 4.1. Unicast - Average Latancy vs. Lambda (Pattern 3 - Heavy Traffic) 8×8×8 Torus



CHAPTER 5 -- MULTICASTS ON TORUS NETWORKS

ATM traffic frequently includes multicasts. CATV and Video conferencing are examples of services that require frequent use of multicasts [30]. Existing packet routing algorithms for the torus networks cannot handle multicasts efficiently [5], [10], [25], [27]. The minimal-fully-adaptive algorithm by Cypher and Gravano [5] is not an exception. It is specifically designed for unicasts. Multicasts algorithms exist for wormhole routing, but are neither suitable nor applicable to packet switching. In this chapter, three multicast algorithms are presented.

5.1 MULTICAST NOTATION

Define a multicast packet as a packet which includes the multicast operator in its destination; for example, in

$$(a_{d-1},\ldots,a_{i+1},*,a_{i-1},\ldots,a_0).$$

'*' is the multicast operator indicating multicast on dimension i. (2, *) on an 8×9 torus network is a multicast to (2,0), (2,1), (2,3), (2,4), (2,5), (2,6), (2,7), and (2,8). On the same network, broadcast can be specified by (*,*). With this notation, it is hard to multicast to a set of arbitrary chosen nodes. To multicast to a set of arbitrary chosen nodes, a multicast or a broadcast, with a message content which selects the arbitrary chosen nodes, is sent first. Then only the chosen nodes will act upon succeeding multicasts or broadcasts while others ignore them. This continues until another multicast or broadcast terminates this mode of operation.

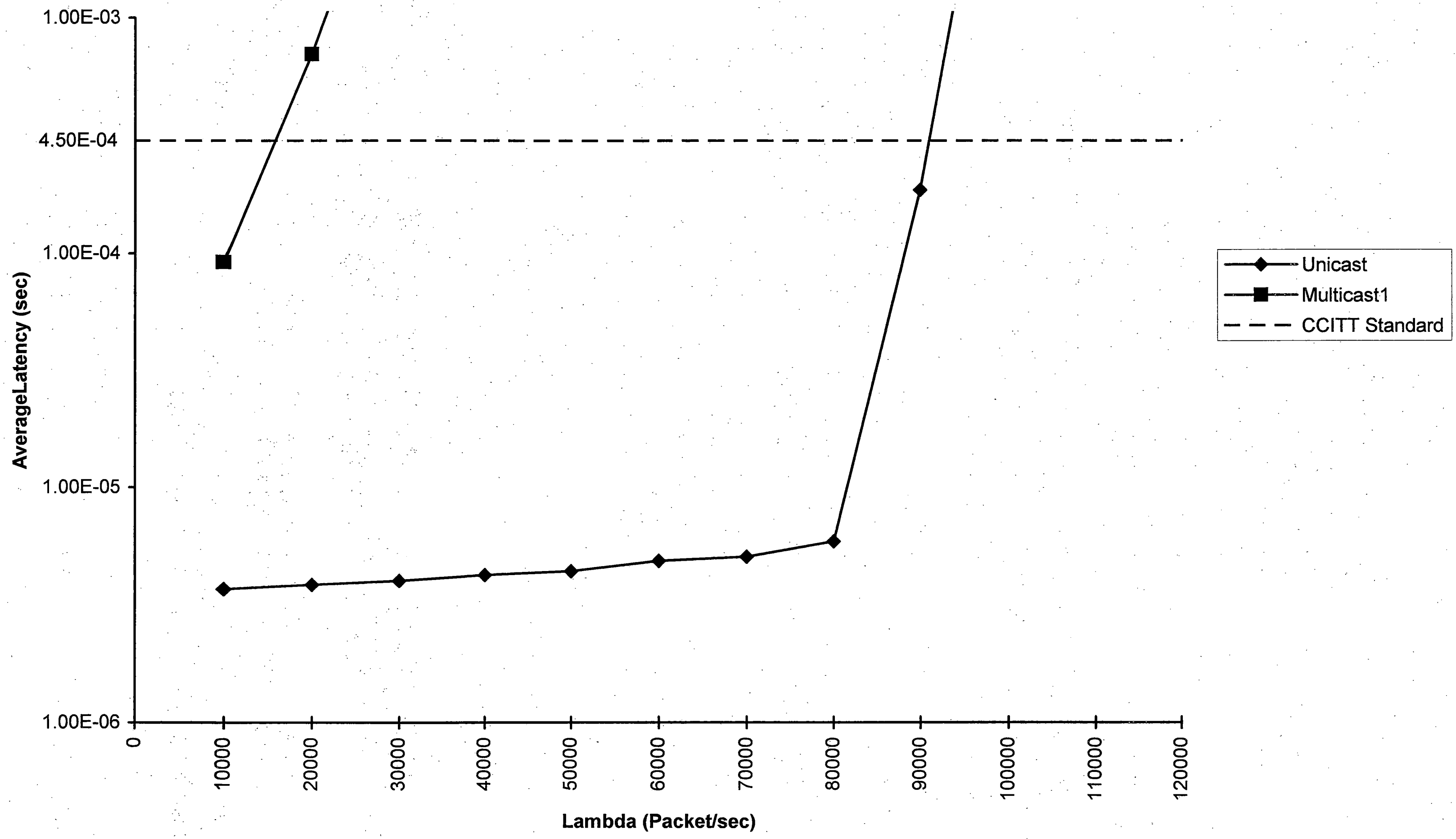
5.2 MULTICAST ALGORITHM 1 - SIMPLISTIC

One way to accomplish a multicast is to send multiple unicasts. The process of sending unicasts from a source node is completely sequential. This implies extra latencies, and more traffic on the network. To accomplish multicasts by multiple unicasts, it is not necessary to modify the unicast algorithm or the queue structure on each node. A multicast packet generates all of its corresponding unicast packets sequentially while at the front of the injection queue.

5.3 SIMULATION RESULT OF MULTICAST ALGORITHM 1

Graph 5.1 shows the simulation result for Multicast Algorithm 1 on pattern 3 (heavy traffic). The network size is 8×8×8. 30% of the packets are multicast packets. They are randomly generated with random target planes. A target plane is a n-dimensional plane if a destination contains n multicast operators where $0 \leq n \leq d-1$. For example, a target plane is a line if a destination contains one multicast operator. Every node on a target plane receives a copy of packet from its source node. Since the simulation of multicasts took too long for higher percent of multicasts, 30% multicasts was selected. However, 30% and 50% multicasts were simulated and their results are shown in Section 5.10. The graph clearly indicates that Multicast Algorithm 1 performs poorly. With 20,000 packets per second mean arrival rate, network latency is already above the CCITT standard. Therefore, it is necessary to investigate more efficient multicast algorithms.

Graph 5.1. Unicast and Multicast1 - Average Latency vs. Lambda
(Pattern 3 - Heavy Traffic)
30% Multicasts on 8×8×8 Torus



5.4 MULTICAST ALGORITHM 2 - RE-INJECTION

The second multicast algorithm tries to reduce network latencies when compared with Multicast Algorithm 1. The inefficiency of Multicast Algorithm 1 is in its sequential generation of unicasts at the source node to perform multicasts. This algorithm handles multicasts more efficiently by re-injecting multicast packets into the injection queue. There is no change in the queue structure of the nodes except for the possibility of inserting a packet from the C queue to the injection queue as in Figure 5.1.

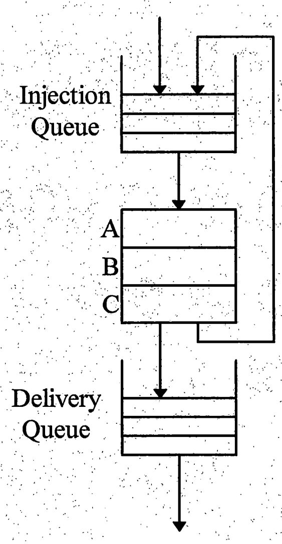


Figure 5.1. The Queue Structure of Multicast Algorithm 2.

Similar to the unicast algorithm, packets enter the torus network by being placed in the injection queue and leave the network from the delivery queue. Routing of a multicast packet consists of two parts, *adaptive unicast* and *distribution*. Multicast packets like unicast packets go through a minimally adaptive route to get to one of the nodes in the 1-, 2-, 3-, etc. dimensional target plane. This is the adaptive unicast part of the algorithm. Once on the target plane, the packet is distributed along dimension $i(0 \le i < d)$, then each node distributes the packet along the next dimension if necessary. This process continues until all desired nodes of the multicast are reached. This process is the distribution part of the algorithm. Multicast Algorithm 2 creates much less traffic than Multicast Algorithm 1. Also, the path traversed from a source node to each destination of the multicast is minimal. For example, consider a multicast packet *p* that is routed from source node (4,2) to destination nodes (2,*) in an 8×9 torus (Figure 5.2).

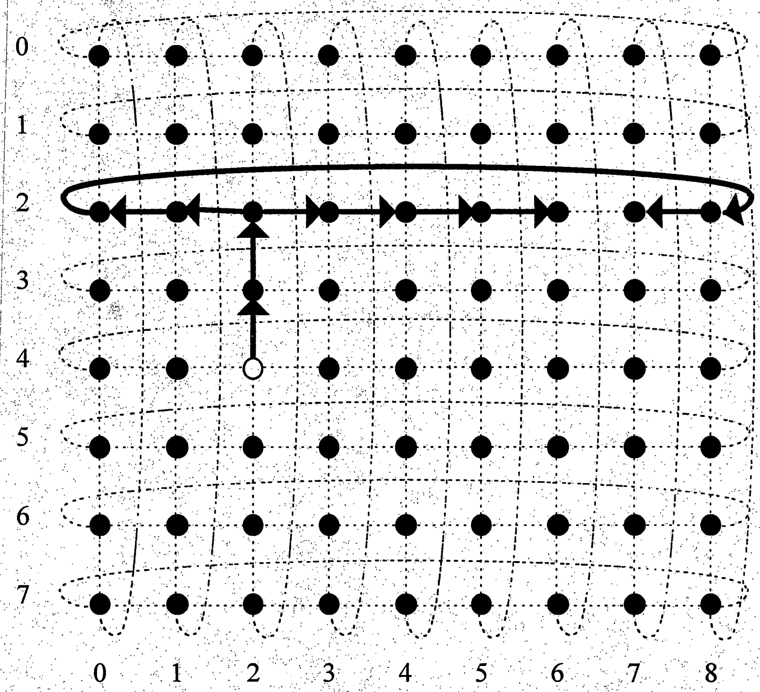


Figure 5.2. An Example of a Multicast Used by Multicast Algorithm 2 in an 8×9 Torus.

The route $(4,2) \rightarrow (3,2) \rightarrow (2,2)$ is the adaptive unicast part. When packet *p* is in node (2,2), the distribution part starts. At this point, two duplicates of packet *p*, packet *q* and packet *r*, are produced. Packet *q*'s destination is set to node (2,6) and packet *r*'s to node (2,7) and are placed in the injection queue of node (2,2). Packet p itself is placed in the delivery queue of the current node (2,2). Since the routings of *q* and *r* are analogous, we concentrate on packet *q*. Starting from the injection queue of node (2,2), packet *q* is routed to node (2,3). From node (2,3) packet *q* moves to node (2,4), but at this time, node (2,3) creates a duplicate of packet *q*. This duplicated packet is eventually routed to the delivery queue of node (2,3). After passing through node (2,5) and being copied by node (2,5), packet *q* will arrive at its destination node (2,6) and move to the delivery queue of node (2,6).

Figure 5.3 is an example of a broadcast, a packet with destination (*,*), on an 8×9 torus. From the source node (6,2), four duplicate packets are re-injected with destinations (*,6), (*,7), (2,2), and (1,2). While the packet with destination (*,6) is being routed, nodes (6,3), (6,4),(6,5), and (6,6) produce two copies with destinations in the next dimension and re-inject them in its injection queues. Each node except for (6,6) passes the packet to the next node while eventually placing a copy in its delivery queue. Node (6,6) just places the packet in its delivery queue. For example, node (6,5) receives a packet from node (6,4) and places a duplicate packet with destination (2,5) and a duplicate packet with destination (1,5) in its injection queue. Node (6,5) also passes a copy to node (6,6) and moves the packet towards its delivery queue.

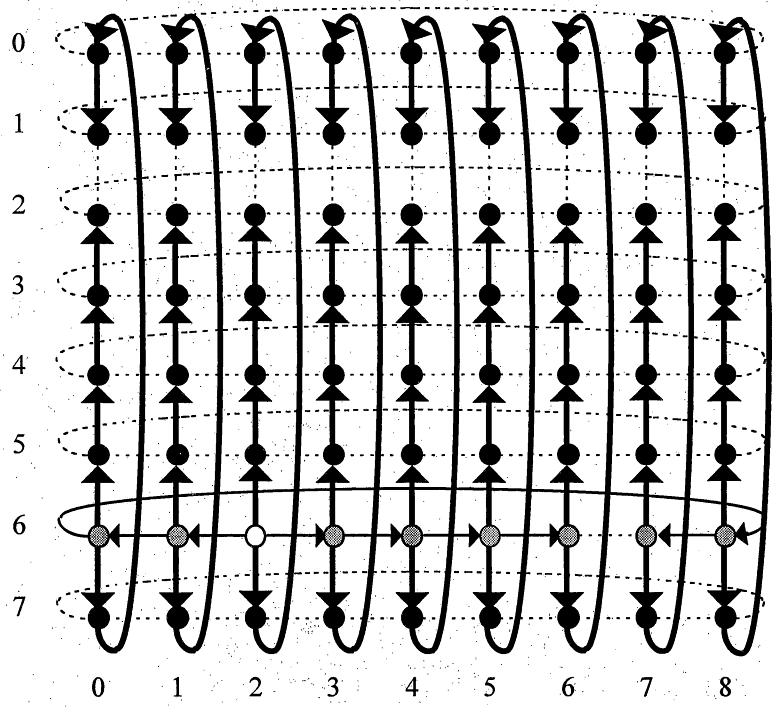


Figure 5.3 An Example of a Broadcast Used by Multicast Algorithm 2 in an 8×9 Torus.

In order to design Multicast Algorithm 2, the calculation of *ok_nodes(p)* must be redefined. For the unicast algorithm, *ok_nodes(p)* is a set of neighboring nodes that lie along a minimal length path to the destination. For Multicast Algorithm 2, we will try to find *ok_nodes(p)* by removing the multicast operator '*' from the mixed radix representation of the node labels. The following is the algorithm to create a temporary destination node label to find *ok_nodes(p)*.

$\forall a_i (0 \leq i < d-1)$ *in destination node* $(a_{d-1}, \ldots, a_i, \ldots, a_0)$ *such* *that* $a_i = '*'$, *replace* a_i *with* a_i' *from the current node* $(a_{d-1}', \ldots, a_i', \ldots, a_0')$.

For example, if a packet *p* is currently in node(6,4) and its destination is node(*,6), the temporary destination will be node(6,6). Now, *ok_nodes(p)* can be found from the temporary node label as in the unicast algorithm. By introducing a special flag *direction*, a subset of *ok_nodes(p)*, called the *allowed_nodes(p)*, will be calculated. The *allowed_nodes(p)* based on the *direction* is as follows:

let $x = node(p)$ and $y \in ok\_nodes(p)$,

If *direction* = ALL Then
$allowed\_nodes(p) = ok\_nodes(p)$
Else if *direction* = POSITIVE Then
$allowed\_nodes(p) = \{y \mid Right(x) < Right(y)\}$
Else if *direction* = NEGATIVE Then
$allowed\_nodes(p) = \{y \mid Left(x) < Left(y)\}$

Similar to *ok_queues(p)*, *allowed_queues(p)* is defined as a set of central queues *in allowed_nodes(p)* that are directly accessible from *node(p)*. A formal description of Multicast Algorithm 2 is given below.

Multicast Algorithm 2

Let A, B, and C be three central queues required by the algorithm (Figure 5.1). Let *p* be an arbitrary packet that is being routed by the algorithm. Let *q* = *queue(p)*, and *x* = *node(p)*. Two flags, *direction* and *distribution*, are used. When packets are inserted to the injection queue, for both unicast and multicast packets, the *distribution* flag is set to NO. The *direction* flag is set to ALL for both types of packets initially. The *distribution* flag can be set to NO, COPY, or PASS to control the duplication of packets on each node. When *distribution* = NO, *p* is either a unicast packet or a multicast packet in the adaptive unicast phase. When *distribution* = COPY, *p* is in the distribution phase of the multicast, and it is required to make a duplicate packet. When *distribution* = PASS, *p* is in the distribution phase of the multicast, and it is not necessary to make a duplicate packet.

During the distribution phase of the multicast, the following sub-tasks become necessary.

Duplicate: Send a copy of *p* to the next node. Change destination(*p*) as follows.

$\forall a_i (0 \le i < d-1)$ in destination node $(a_{d-1},\ldots,a_i,\ldots,a_0)$ such that $a_i \neq '*'$, replace a_i with a_i' from the current node $(a_{d-1}',\ldots,a_i',\ldots,a_0')$.

Change_Flags : Change *direction* of *p* to ALL and set *distribution* to COPY before *x* sends *p* to the next node.

Multi_Duplicate: When *p* moves to the delivery queue, do the following.

For $i = 0$ to $d-1$ Do

If $a_i =$ '*' where a_i is in destination node $(a_{d-1},\ldots,a_i,\ldots,a_0)$ then

- put a duplicate of *p* in the injection queue with a new destination, *direction*, and *distribution* as follows.

  $tmp = b_i - \lfloor k_i/2 \rfloor$ where b_i is in current node $(b_{d-1},\ldots,b_i,\ldots,b_0)$

  If $tmp \ge 0$ then

  $a_i = tmp$

  Else

  $a_i = tmp + k_i$

  End If

  direction = NEGATIVE

  distribution = PASS

- put second duplicate of *p* in the injection queue with a new destination, *direction*, and *distribution* as follows.

  If $k_i \bmod 2 = 0$ Then

  $a_i = (b_i + \lfloor k_i/2 \rfloor - 1) \bmod k_i$

  Else

  $a_i = (b_i + \lfloor k_i/2 \rfloor) \bmod k_i$

  End If

direction = POSITIVE
distribution = PASS

- $a_i = b_i$

End For

The algorithm creates *p*'s waiting set *wait(p)* based on the following cases.

Case 1: *q* is an injection queue.
In this case, *wait(p)* consists of the A queue in x.
Case 2: *q* is an *A* queue.
In this case, there are two subcases.

Case 2a: $\exists y \in allowed\_nodes(p)$ **such that** $Right(x) < Right(y)$.
In this subcase, *wait(p)* consists of all of the *A* queues in *allowed_queues(p)*.
If *distribution* = PASS Then
  Perform Change_Flags
If *distribution* = COPY Then
  Perform Duplicate
End If

Case 2b: $\nexists y \in allowed\_nodes(p)$ **such that** $Right(x) < Right(y)$.
In this subcase, *wait(p)* consists of the *B* queue in x.
Case 3: *q* is a *B* queue.
In this case, there are two subcases.

Case 3a: $\exists y \in allowed\_nodes(p)$ **such that** $Left(x) < Left(y)$.
In this subcase, *wait(p)* consists of all of the *B* queues in *allowed_queues(p)*.
If *distribution* = PASS Then
  Perform Change_Flags
If *distribution* = COPY Then
  Perform Duplicate
End If

Case 3b: $\nexists y \in allowed\_nodes(p)$ **such that** $Left(x) < Left(y)$.
In this subcase, *wait(p)* consists of the *C* queue in x.
Case 4: *q* is a *C* queue.
In this case, there are three subcases.

Case 4a: $x \neq destination(p)$ AND $|$*allowed_nodes(p)*$| \neq 0$.
In this subcase, *wait(p)* consists of all of the *C* queues in *allowed_queues(p)*.
If *distribution* = PASS Then
  Perform Change_Flags
Else If *distribution* = COPY Then
  Perform Duplicate
End If

Case 4b: $x \neq destination(p)$ AND $|$*allowed_nodes(p)*$| = 0$

In this subcase, *wait(p)* consists of the delivery queue in x. Perform Multi_Dupulicate

Case 4c: $x = destination(p)$.

In this subcase, *wait(p)* consists of the delivery queue in x.

Case 5: *q* is a delivery queue.

5.5 PROOF OF CORRECTNESS FOR MULTICAST ALGORITHM 2

In this section, freedom from deadlock, livelock, and, starvation is shown for Multicast Algorithm 2. Since the queue structure of Multicast Algorithm 2 is not changed from the unicast algorithm, it is immediate that it is free from deadlock, livelock, and starvation for unicasts.

Definition: Let *q* be any queue in the torus network that is used by Multicast Algorithm 2, and let x denote the node in which *q* is located and *n* denote the nodes in the torus network. The ranking function Rank1(*q*) is defined as follows.

$$Rank1(q) = \begin{cases} Right(x) & \text{if } q \text{ is an injection queue} \\ n + Right(x) & \text{if } q \text{ is an } A \text{ queue} \\ 2n + Left(x) & \text{if } q \text{ is a } B \text{ queue} \\ 3n + Inside(x) & \text{if } q \text{ is a } C \text{ queue} \\ 4n + Right(x) & \text{if } q \text{ is a delivery queue} \end{cases}$$

The following lemma, due to Cypher and Gravano, still holds for Multicast Algorithm 2.

Lemma 5.1 (Cypher and Gravano): Let *p* be any packet that is being routed by Multicast Algorithm 2 and let *q* = *queue(q)*. Either *q* is the delivery queue in *destination(p)* or there exists a queue $w \in wait(p)$ such that *Rank1(q)* < *Rank1(w)*.

The following lemma proves that Multicast Algorithm 2 is free of livelock.

Lemma 5.2: If p is any multicast packet that is being routed by Multicast Algorithm 2, then p will be stored in at most a finite number of queues before being placed in the delivery queue of its destination nodes.

Proof: Because p always takes a minimal length path to all its destinations, it visits only a finite number of nodes. When p finishes the adaptive part of the multicast algorithm, it is sent to the delivery queue of the current node and two duplicate packets p' and p'' are put into the injection queue of the current node for each dimension i of the multicast. Whenever, p, p', or p'' visit a node, they are stored in each injection, A, B, C, and delivery queue at most once because the multicast algorithm visits each queue type in monotonically increasing order. □

To finish the proof for Multicast Algorithm 2, there is one assumption that needs to be made. Since Multicast Algorithm 2 re-feeds duplicate packets from the C queue into the injection queue, the injection queue needs to be large enough not to cause deadlock. In the worst case, the injection queue can be filled and deadlock can happen. However, because of the simulation result in the next section, a large enough queue size can be chosen to prevent deadlocks.

Theorem 5.3: Multicast Algorithm 2 is free of deadlock, livelock, and starvation.

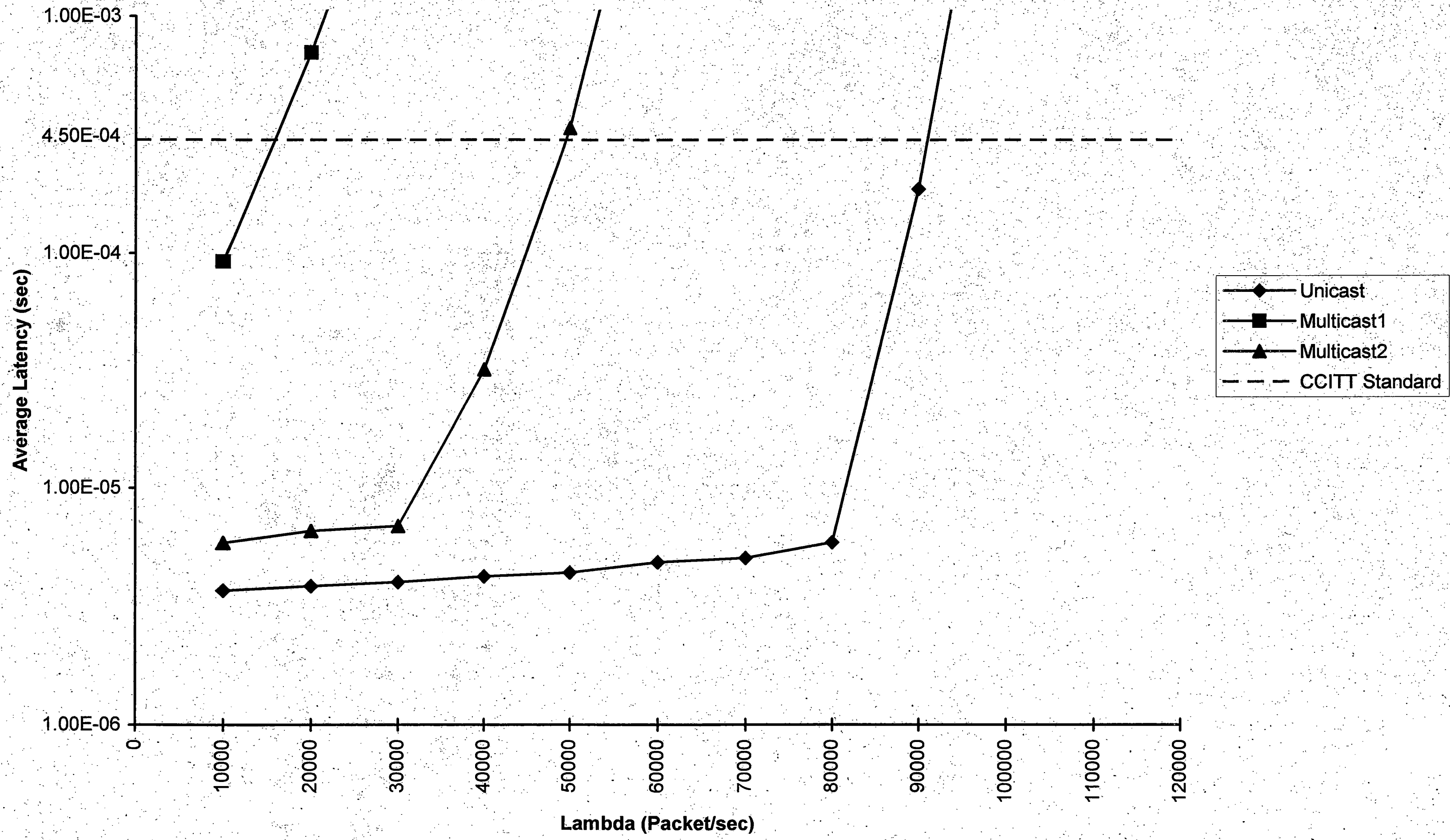
Proof:

- Deadlock Free - from Lemmas 3.1 and 5.1, and the assumption above Multicast Algorithm 2 can be prevented from deadlock.
- Starvation Free - from Lemmas 3.1 and 5.1, it follows that once a packet has been placed in an injection queue, it never remains in a single queue forever, Lemma 3.1. Therefore, Multicast Algorithm 2 is free of starvation.
- Livelock Free - from Lemma 5.2 and the fact that no single packet remains in a single queue forever, every packet will eventually arrive at the delivery queue of its destinations. Therefore Multicast Algorithm 2 is free of livelock. □

5.6 SIMULATION RESULT OF MULTICAST ALGORITHM 2

Graph 5.2 shows the simulation result of Multicast Algorithm 2 with the unicast algorithm and Multicast Algorithm 1. Pattern 3 (heavy traffic) with 30% multicast packets is used. The network size is 8×8×8. Multicast Algorithm 2 shows significant improvement over Multicast Algorithm 1. By simply re-injecting multicast packets to the injection queue, Multicast Algorithm 2 can handle multicasts much more efficiently. Table 5.1 shows other results of the simulation. It is important to note the maximum injection queue size and the average injection queue size. When the average latency exceeds 1 second, the maximum injection queue size is 279 and the average injection queue size is 191.980. With the maximum injection queue size of 279 (279×57 bytes), 1 MByte is more than sufficient to prevent deadlocks. 1 MByte with current technology is a very reasonable queue size. Therefore, Multicast Algorithm 2 requires only reasonably sized injection queues.

Graph 5.2. Unicast, Multicast1, and Multicast2 - Average Latency vs. Lambda (Pattern 3 - Heavy Traffic) 30% Multicasts on 8×8×8 Torus



λ	Network Throughput (bps)	Avg. Latency (sec)	Max. Injection Queue Size	Avg. Injection Queue Size	Max. Delivery Queue Size	Avg. Delivery Queue Size	Avg. Queue Utilization (%)
10000	1.38E+10	5.85E-06	6	1.455	7	1.357	9.458
20000	2.52E+10	6.56E-06	5	1.473	14	2.270	17.952
30000	3.19E+10	6.89E-06	5	1.485	44	6.296	24.084
40000	4.36E+10	3.21E-05	105	15.504	105	26.963	50.082
50000	3.68E+10	3.39E-04	193	104.573	53	6.068	69.260
60000	2.73E+10	1.00E-02	223	145.932	32	2.776	85.527
70000	2.05E+10	3.31E-01	270	174.515	12	1.930	88.919
80000	1.36E+10	1.51E+01	279	191.980	11	1.539	93.874

Table 5.1. Multicast2 - Pattern 3 (Heavy Traffic)
Network Size: 8×8×8, 30% Multicasts

5.7 MULTICAST ALGORITHM 3 - MULTIPLE CENTRAL QUEUES

Although Multicast Algorithm 2 handles multicasts much more efficiently than Multicast Algorithm 1, congestion in the A, B, and C queues caused by the re-injection of packets quickly slows down the algorithm. Unicast packets may be unnecessarily delayed. Multicast algorithm 3 handles multicasts in a separate set of queues, D, E, and F as in Figure 5.4.

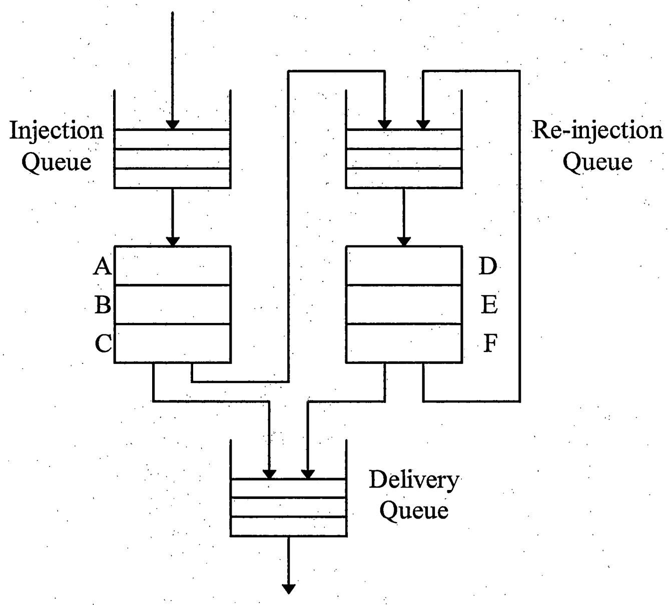


Figure 5.4. The Queue Structure of Multicast Algorithm 3.

An additional queue, called *re-injection queue*, is introduced. In Multicast Algorithm 3, multicast packets are duplicated in the C , D, E, or F queues and placed in the re-injection queue. Multicast packets in the re-injection queue will move to the D queue to perform multicasts. By handling multicasts in separate queues, unicast packets will not be delayed unnecessarily. Similar to Multicast Algorithm 2, unicast packets are handled as in the unicast algorithm. *allowed_nodes(p)* and *allowed_queues(p)* are generated as in Multicast Algorithm 2. Multicast Algorithm 3 uses the same partially adaptive routing method as Multicast Algorithm 2. After a multicast packet reaches its target plane, Multicast Algorithm 2 places duplicate packets into the injection queue of the current node while Multicast Algorithm 3 places duplicate packets into the re-injection queue of the current node. Therefore, the distribution part of the algorithm is completely separated from the adaptive part of the algorithm. For example, consider a multicast packet *p* that is being routed from source node (4,2) to destination nodes (2,*) in an 8×9 torus (Figure 5.2). While *p* is on nodes (4,2), (3,2), and (2,2), the adaptive unicast part of Multicast Algorithm 3 is performed and *p* is stored in the injection, A, B, or C queue of these nodes. Once *p* has reached node (2,2), two duplicate packets of *p* will be created and stored in the re-injection queue of node (2,2). Thereafter, these copied packets of *p* will be handled only in the re-injection, D, E, F, and delivery queues. The following is the formal definition of Multicast Algorithm 3.

Multicast Algorithm 3

Let A, B, C, D, E, and F be six central queues required by the algorithm (Figure 5.4). Let *p* be an arbitrary packet that is being routed by the algorithm. Let *q* = *queue(p)*, and *x* = *node(p)*. Two flags, *direction* and *distribution*, are used. When packets are inserted to the injection queue, for both unicast and multicast packets, the *distribution* flag is set to NO. The *direction* flag is set to ALL for both types of packets initially. The *distribution* flag can be set to NO, COPY, or PASS to control the duplication of packets on each node. When *distribution* = NO, *p* is either a unicast packet or a multicast packet in the adaptive unicast phase. When *distribution* = COPY, *p* is in the distribution phase of the multicast, and it is required to make a duplicate packet. When *distribution* = PASS, *p* is in the distribution phase of the multicast, and it is not necessary to make a duplicate packet.

During the distribution phase of the multicast, the following sub-tasks become necessary.

Duplicate: Send a copy of *p* to the next node. Change destination(*p*) as follows.

$\forall a_i (0 \le i < d-1)$ in destination node $(a_{d-1},\ldots,a_i,\ldots,a_0)$ such that $a_i \ne '*'$, replace a_i with a_i' from the current node $(a_{d-1}',\ldots,a_i',\ldots,a_0')$.

Change_Flags : Change *direction* of *p* to ALL and set *distribution* to COPY before *x* sends *p* to the next node.

Multi_Duplicate: When *p* moves to the delivery queue, do the following.

For $i = 0$ to $d-1$ Do
  If a_i = '*' where a_i is in destination node $(a_{d-1},\ldots,a_i,\ldots,a_0)$ then

- put a duplicate of *p* in the injection queue with a new destination, *direction*, and *distribution* as follows.

  $tmp = b_i - \lfloor k_i / 2 \rfloor$ where b_i is in current node $(b_{d-1},\ldots,b_i,\ldots,b_0)$

  If $tmp \ge 0$ then

  $a_i = tmp$

  Else

  $a_i = tmp + k_i$

  End If

  direction = NEGATIVE

  distribution = PASS

- put second duplicate of *p* in the injection queue with a new destination, *direction*, and *distribution* as follows.

  If $k_i \bmod 2 = 0$ Then

  $$a_i = (b_i + \lfloor k_i / 2 \rfloor - 1) \bmod k_i$$

  Else

  $$a_i = (b_i + \lfloor k_i / 2 \rfloor) \bmod k_i$$

  End If
  direction = POSITIVE
  distribution = PASS
- $a_i = b_i$

End For

The algorithm creates *p's* waiting set *wait(p)* based on the following cases.

Case 1: *q* is an injection queue.
In this case, *wait(p)* consists of the A queue in x.
Case 2: *q* is an *A* queue.
In this case, there are two subcases.

Case 2a: $\exists y \in allowed\_nodes(p)$ **such that** $Right(x) < Right(y)$.
In this subcase, *wait(p)* consists of all of the *A* queues in *allowed_queues(p)*.

Case 2b: $\nexists y \in allowed\_nodes(p)$ **such that** $Right(x) < Right(y)$.
In this subcase, *wait(p)* consists of the *B* queue in x.

Case 3: *q* is a *B* queue.
In this case, there are two subcases.

Case 3a: $\exists y \in allowed\_nodes(p)$ **such that** $Left(x) < Left(y)$.
In this subcase, *wait(p)* consists of all of the *B* queues in *allowed_queues(p)*.

Case 3b: $\nexists y \in allowed\_nodes(p)$ **such that** $Left(x) < Left(y)$.
In this subcase, *wait(p)* consists of the *C* queue in x.

Case 4: *q* is a *C* queue.
In this case, there are three subcases.

Case 4a: $x \neq destination(p)$ AND $|allowed\_nodes(p)| \neq 0$.
In this subcase, *wait(p)* consists of all of the *C* queues in *allowed_queues(p)*.

Case 4b: $x \neq destination(p)$ and $|allowed\_nodes(p)| = 0$.
In this subcase, *wait(p)* consists of the delivery queue in x. Perform Multi_Dupulicate.

Case 4c: $x = destination(p)$.
In this subcase, *wait(p)* consists of the delivery

queue in x.

Case 5: *q* is a *D* queue.

In this case, there are two subcases.

Case 5a: $\exists y \in allowed\_nodes(p)$ **such that** $Inside(x) < Inside(y)$.

In this subcase, *wait(p)* consists of all of the *D* queues in *allowed_queues(p)*.

```
If distribution = PASS Then
  Perform Change_Flags
Else If distribution = COPY Then
  Perform Duplicate
End If
```

Case 5b: $\nexists y \in allowed\_nodes(p)$ **such that** $Inside(x) < Inside(y)$.

In this subcase, *wait(p)* consists of the *E* queue in x.

Case 6: *q* is a *E* queue.

In this case, there are two subcases.

Case 6a: $\exists y \in allowed\_nodes(p)$ **such that** $Outside(x) < Outside(y)$.

In this subcase, *wait(p)* consists of all of the *E* queues in *allowed_queues(p)*.

```
If distribution = PASS Then
  Perform Change_Flags
Else If distribution = COPY Then
  Perform Dupulicate
End If
```

Case 6b: $\nexists y \in allowed\_nodes(p)$ **such that** $Outside(x) < Outside(y)$.

In this subcase, *wait(p)* consists of the *F* queue in x.

In this case, there are two subcases.

Case 7: *q* is a *F* queue.

In this case, there are three subcases.

Case 7a: $x \neq destination(p)$ AND $|allowed\_nodes(p)| \neq 0$.

In this subcase, *wait(p)* consists of all of the *F* queues in *allowed_queues(p)*.

```
If distribution = PASS Then
  Perform Change_Flags
Else If distribution = COPY Then
  Perform Duplicate
End If
```

Case 7b: $x \neq destination(p)$ and $|allowed\_nodes(p)| = 0$.

In this subcase, *wait(p)* consists of the delivery queue in x. Perform Multi_Duplicate.

Case 7c: $x = destination(p)$.
In this subcase, *wait(p)* consists of the delivery queue in x.
Case 8: q **is a Re-injection queue.**
In this case, *wait(p)* consists of the D queue in x.
Case 9: q **is a delivery queue.**

5.8 PROOF OF CORRECTNESS FOR MULTICAST ALGORITHM 3

Similarly to Multicast Algorithm 2, unicast packets are routed based on the unicast algorithm. To prove Multicast Algorithm 3 is free of deadlock and starvation, the total ordering of queues in the torus has to be defined.

The following lemma [5] is used to prove that packets that are stored in C queues only move to the inside. This lemma is essential to prove that Multicast Algorithm 3 is free from deadlock and has been proved.

Lemma 5.4 (Cypher and Gravano): Let p be any packet that is being routed by the algorithm, and let $(a_{d-1}, a_{d-2}, \ldots, a_0)$ denote the address of node(p). If queue(p) is a C queue, then for each dimension i, $(0 \le i < d)$, either p requires no further moves or along dimension i or p's next move along dimension i will occur inside.

The following lemma shows that packets that are stored in F queues only moves to the inside. This fact will be important to prove that Multicast Algorithm 3 is free from deadlock along with Lemma 5.4.

Lemma 5.5: Let p be any packet that is being routed by the algorithm, and let $(a_{d-1}, a_{d-2}, \ldots, a_0)$ denote the address of node(p). If queue(p) is an F queue, then for each dimension i, $(0 \le i < d)$, either p requires no further moves along dimension i or p's next move along dimension i will occur inside.

Proof: For each multicast operation on dimension i, Multicast Algorithm 3 creates two duplicate packets. These two copied packets are required to traverse at most $\lfloor k_i/2 \rfloor$ hops. Since any duplicate packet needs to be routed on the dimension of multicast operation only, we can concentrate on a 1-dimensional torus. Let s be the node on which two duplicate packets are created. Consider 5 cases.

Case 1: $s = \lfloor k_i/2 \rfloor$

Packets in both the positive and negative directions need to move to the E queue. When they finish traversing the distance of $\lfloor k_i/2 \rfloor$, then they are in the E queue. Therefore, in the F queue, they require no further movement.

Case 2: $s = 0$.

In this case, packets in both the positive and negative directions finish traversing the distance of $\lfloor k_i/2 \rfloor$ while they are in the D queues. Therefore, when they reach the F queues, they require no further movement.

Case 3: $s = k_i - 1$.

In this case, a packet in the negative direction stays in the D queues to move the length of $\lfloor k_i / 2 \rfloor$, and in the F queue, it requires no further movement. A packet in the positive direction first moves to the E queue of s to move along the wraparound connection. Then it moves to the F queue to move inside only.

Case 4: $0 < s < \lfloor k_i / 2 \rfloor$.

A packet in the positive direction stays in the D queues until node $\lfloor k_i / 2 \rfloor$. At node $\lfloor k_i / 2 \rfloor$, it moves to the E queue to move outside. When the packet reaches the F queue, it requires no further movement. A packet in the negative direction first moves to the E queue of s in order to move in the negative direction. It stays in the E queues until the wraparound connection. To move along the wraparound connection, it moves to the F queue. Thereafter, it only moves inside.

Case 5: $\lfloor k_i / 2 \rfloor < s < k_i - 1$.

A packet in the positive direction first needs to move to the E queue of s so that it can move in the positive direction. After it moved along the wraparound connection, it moves to the F queue to go inside only.

A packet in the negative direction stays in the D queues until it reaches node $\lfloor k_i/2 \rfloor$. At node $\lfloor k_i/2 \rfloor$, it moves to the E queue to move outside. When the packet reaches the F queue, it requires no further movement. □

Definition: Let q be any queue in the torus network that is used by Multicast Algorithm 3, and let x denote the node in which q is located. Again, n denotes the number of nodes in the torus network. The following function Rank2(q) is defined as follows.

$$Rank2(q) = \begin{cases} Right(x) & \text{if } q \text{ is an injection queue} \\ n + Right(x) & \text{if } q \text{ is an } A \text{ queue} \\ 2n + Left(x) & \text{if } q \text{ is a } B \text{ queue} \\ 3n + Inside(x) & \text{if } q \text{ is a } C \text{ queue} \\ 4n + Right(x) & \text{if } q \text{ is a re-injection queue} \\ 5n + Inside(x) & \text{if } q \text{ is a } D \text{ queue} \\ 6n + Outside(x) & \text{if } q \text{ is an } E \text{ queue} \\ 7n + Inside(x) & \text{if } q \text{ is a } F \text{ queue} \\ 8n + Right(x) & \text{if } q \text{ is a delivery queue} \end{cases}$$

The ranking of injection, A, B, C, and delivery queues are still the same as in the ranking function Rank1(q) of Multicast Algorithm 2. Multicast Algorithm 3 routes unicast packets as in the unicast algorithm and Multicast Algorithm 2. Therefore, for unicast packets, Multicast Algorithm 3 is immediately free of deadlock, livelock, and starvation.

Lemma 5.6: Let p be any packet that is being routed by Multicast Algorithm 3 and let q = *queue(p)*. Either q is the delivery queue in *destination(p)* or there exists a queue $w \in wait(p)$ such that Rank2(q) < Rank2(w).

Proof: Let $x = node(p)$. Consider each of the case of the definition of *wait(P)* separately. Also, remember that when two duplicate packets of p are created for each multicast dimension i in the C or F queue and placed into the re-injection queue of the current node, the original packet p will be moved to the delivery queue of the current node to be removed from the network. Thereafter, the rest of multicasting is carried out by these new duplicate packets.

Case 1: q **is an injection queue.**

In this case, let w be the A queue in x and note that *Rank2(q)* < *Rank(w)*.

Case 2: q **is an** A **queue.**

In this case there are two subcases.

Case 2a: $\exists y \in allowed\_nodes(p)$ **such that** $Right(x) < Right(y)$.

In this subcase, let w be the A queue in y and note that *Rank2(q)* < *Rank2(w)*.

In this subcase, let w be the B queue in x and node that that *Rank2(q)* < *Rank2(w)*.

Case 2b: $\nexists y \in allowed\_nodes(p)$ **such that** $Right(x) < Right(y)$.

Case 3: q **is a** B **queue.**

In this case there are two subcases.

Case 3a: $\exists y \in allowed\_nodes(p)$ **such that** $Left(x) < Left(y)$.

In this subcase, let w be the B queue in y and note that *Rank2(q)* < *Rank2(w)*.

Case 3b: $\nexists y \in allowed\_nodes(p)$ **such that** $Left(x) < Left(y)$.

In this subcase, let w be the C queue in y and note that *Rank2(q)* < *Rank2(w)*.

Case 4: q **is a** C **queue.**

In this case there are three subcases.

Case 4a: $x \neq destination(p)$ AND $|allowed\_nodes(p)| \neq 0$.

In this subcase, let y be any node in *allowed_nodes(p)*. It follows from Lemma 5.4 that $Inside(x) < Inside(y)$, so let w be the C queue in y and note that *Rank2(q)* < *Rank2(w)*.

Case 4b: $x \neq destination(p)$ and $|allowed\_nodes(p)| = 0$.

In this subcase, let w be the delivery queue in x and node that *Rank2(q)* < *Rank2(w)*.

Case 4c: $x = destination(p)$.

In this subcase, let w be the delivery queue in x and node that *Rank2(q)* < *Rank2(w)*.

Case 5: q **is a** D **queue.**

In this case there are two subcases.

Case 5a: $\exists y \in allowed\_nodes(p)$

such that $Inside(x) < Inside(y)$ **.**

In this subcase, let w be the D queue in y and note that *Rank2(q)* < *Rank2(w)*.

Case 5b: $\nexists y \in allowed\_nodes(p)$

such that $Inside(x) < Inside(y)$ **.**

In this subcase, let w be the E queue in x and note that *Rank2(q)* < *Rank2(w)*.

Case 6: q is an E queue.

In this case there are two subcases.

Case 6a: $\exists y \in allowed\_nodes(p)$

such that $Outside(x) < Outside(y)$ **.**

In this subcase, let w be the E queue in y and note that *Rank2(q)* < *Rank2(w)*.

Case 7b: $\nexists y \in allowed\_nodes(p)$

such that $Outside(x) < Outside(y)$ **.**

In this subcase, let w be the F queue in x and note that *Rank2(q)* < *Rank2(w)*.

Case 7: q is an F queue.

In this case there are three subcases.

Case 7a: $x \neq destination(p)$ AND $|$*allowed_nodes(p)*$| \neq 0$**.**

In this subcase, let y be any node in *allowed_nodes(p)*. It follows from Lemma 5.5 that $Inside(x) < Inside(y)$, so let w be the F queue in y and note that *Rank2(q)* < *Rank2(w)*.

Case 7b: $x \neq destination(p)$ and $|allowed\_nodes(p)| = 0$.

In this subcase, let w be the delivery queue in x and node that *Rank2(q)* < *Rank2(w)*.

Case 7c: $x = destination(p)$.

In this subcase, let w be the delivery queue in x and node that *Rank2(q)* < *Rank2(w)*.

Case 8: q **is a re-injection queue.**

In this case let w be the D queue in x and note that *Rank2(x)* < *Rank2(w)*.

Case 9: q **is a delivery queue.**

In this case, the lemma holds trivially. □

To finish the proof for Multicast Algorithm 3, there is one assumption that we need to make as we did for Multicast Algorithm 2. Since Multicast Algorithm 3 re-feeds duplicate packets from the F queue into the re-injection queue, the re-injection queue needs to be large enough not to cause deadlock. This assumption becomes reasonable when we study the simulation result in the next section, and it is possible to choose a large enough queue size.

Theorem 5.7: Multicast Algorithm 3 is free of deadlock, livelock, and starvation.

Proof:

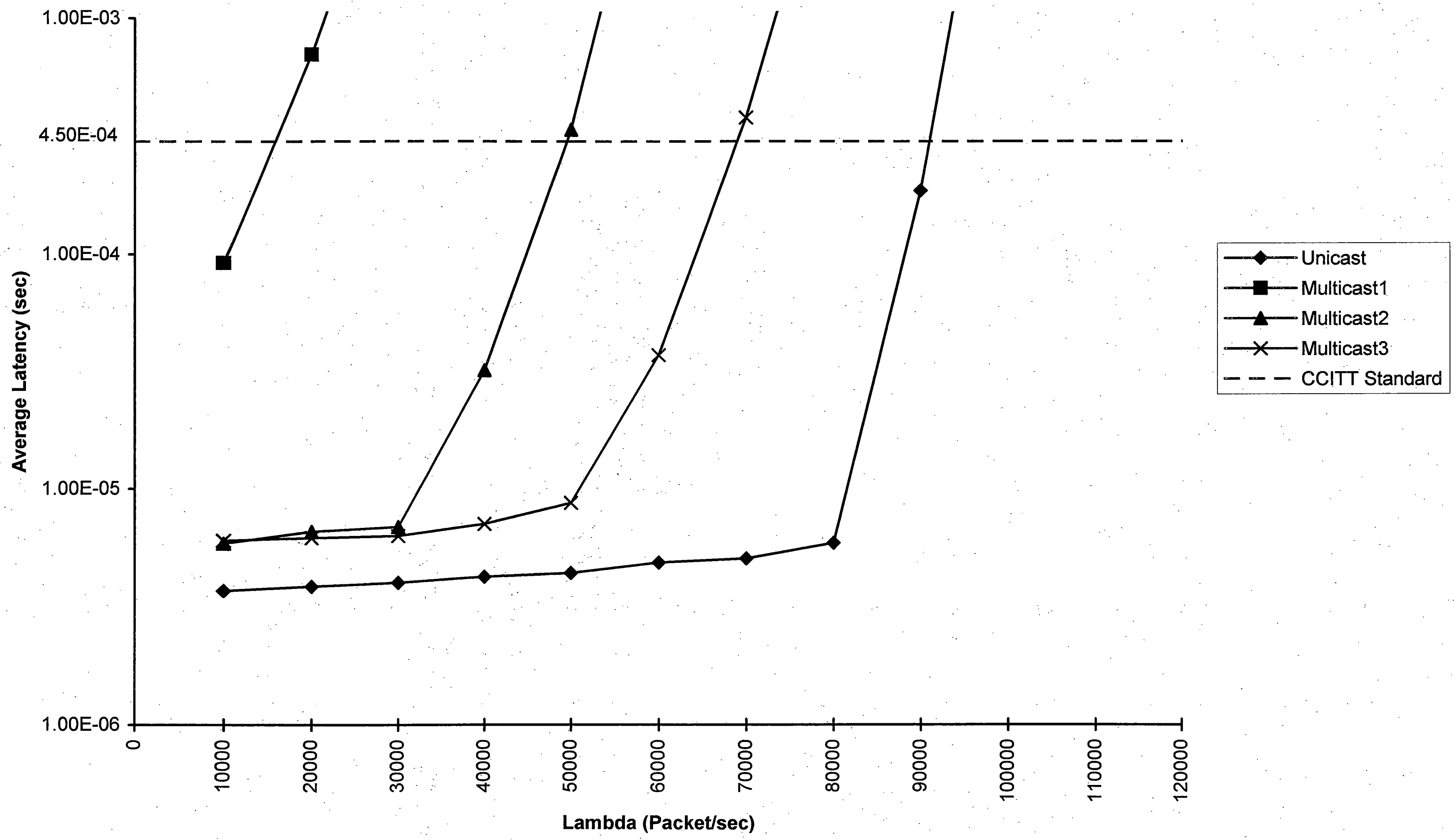
- Deadlock Free - from Lemmas 3.1 and 5.6, and the assumption above, Multicast Algorithm 3 can be prevented from deadlock.

- Starvation Free - from Lemmas 3.1 and 5.6, it follows that once a packet has been placed in an injection queue, it never remains in a single queue forever, Lemma 3.1. Therefore, Multicast Algorithm 3 is free of starvation.
- Livelock Free - from Lemma 5.2 and the fact that no single packet remains in a single queue forever, every packet will eventually arrive at the delivery queue of its destinations. Therefore Multicast Algorithm 3 is free of livelock. □

5.9 SIMULATION RESULT OF MULTICAST ALGORITHM 3

Graph 5.3 indicates simulation results of Multicast Algorithm 3 on an 8×8×8 torus network with the results of the other algorithms. The latency curve of Multicast Algorithm 3 is much closer to the latency curve of the unicast algorithm. This result clearly indicates that multicast algorithm 3 handles multicasts better than the previous two multicast algorithms. Table 5.2 shows other results of the simulation. The injection queue and the re-injection queue do not grow large. When the average latency exceeds 1 second, the sum of the maximum injection queue size (253 packets) and the maximum re-injection queue size (9 packets) is even smaller than the maximum injection queue size of Multicast Algorithm 2 (279 packets). Multicast Algorithm 3 requires reasonably sized injection and re-injection queues. Also, the size of injection queue is very close to 1 most of the time. This indicates that unicasts packets are not delayed unnecessarily. It is interesting to observe the size of re-injection queue. Once congestion starts on the network, the size of the re-injection queue drops significantly. This result indicates that congestion is mainly occurring in the A, B, and C queues. Two simulation results of three multicast algorithms on a 4×4×4 torus network using pattern 3 are given in Graphs 5.4 and 4.5. In Graph 5.4, multicast packets are 30% of all packets. In Graph 5.5, multicast packets are 50% of all packets. In every case, Multicast Algorithm 3 outperforms Multicast Algorithm 1 and Multicast Algorithm 2.

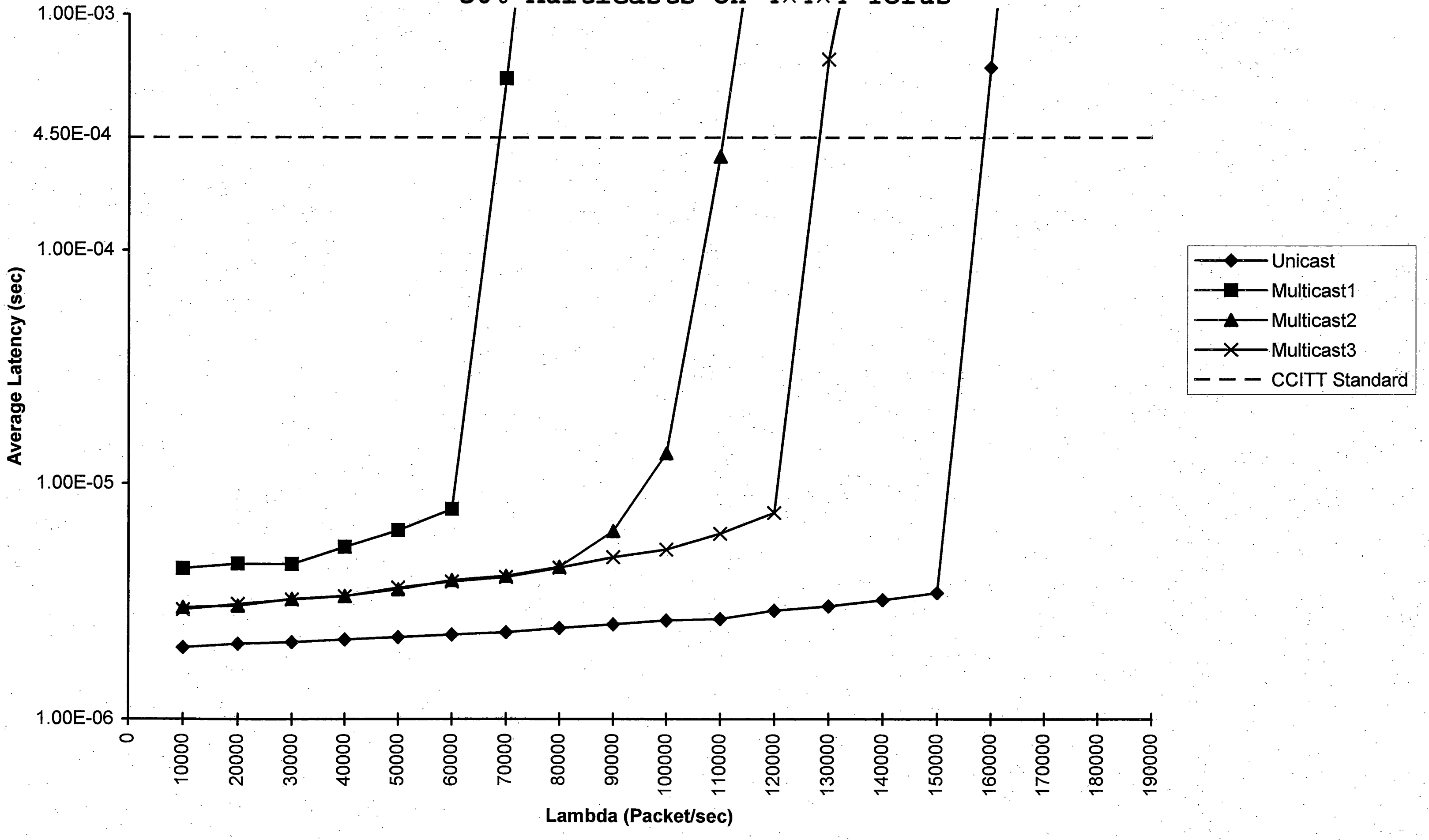
Graph 5.3. Unicast, Multicast1, Multicast2, and Multicast3 - Average Latency vs. Lambda (Pattern 3 - Heavy Traffic) 30% Multicasts on 8×8×8 Torus

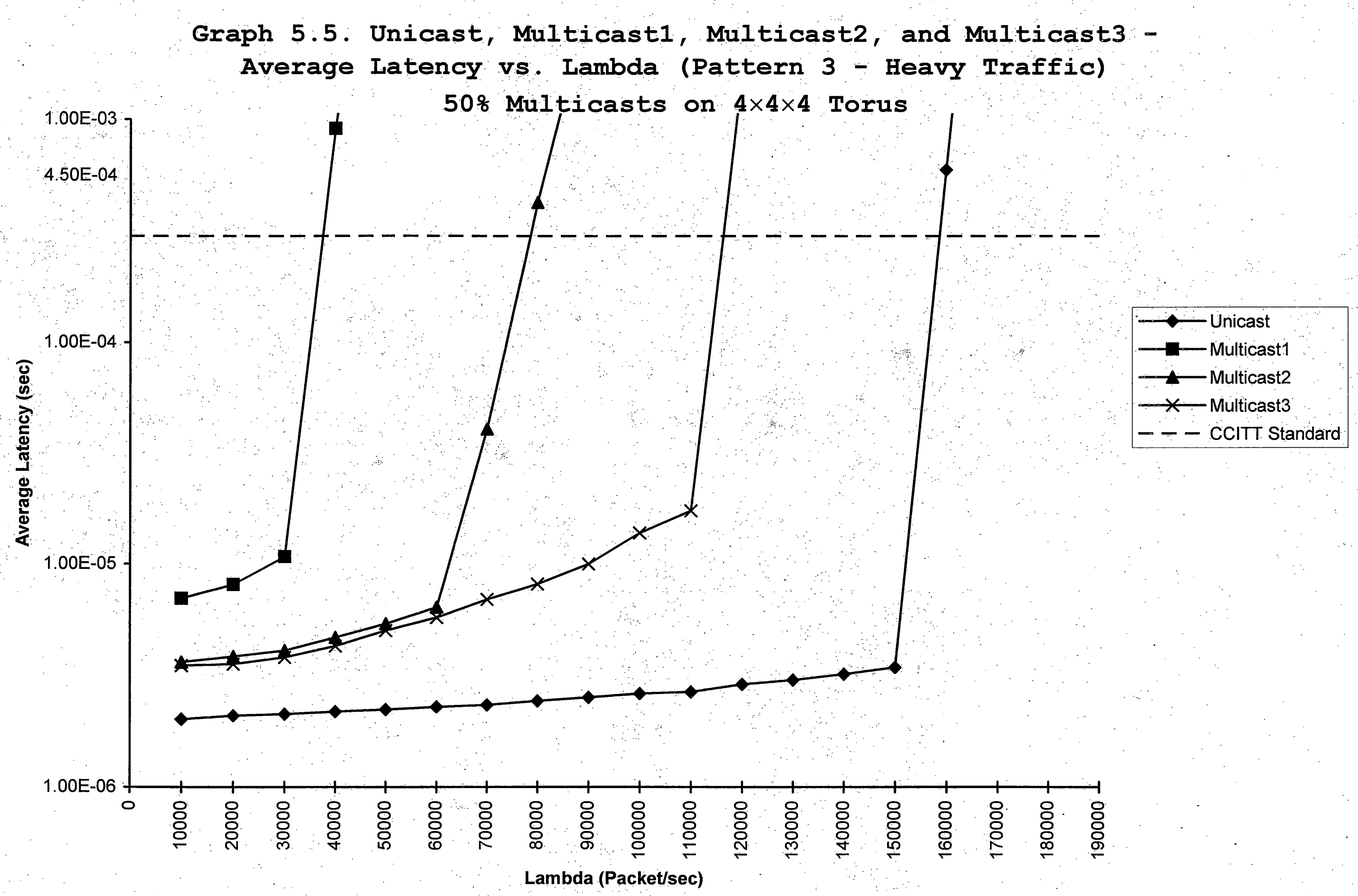


λ	Network Throughput (bps)	Avg. Latency (sec)	Max. Injection Queue Size	Avg. Injection Queue Size	Max. Re-injection Queue Size	Avg. Re-injection Queue Size	Max. Delivery Queue Size	Avg. Delivery Queue Size	Avg. Queue Utiliza-tion (%)
10000	1.61E+10	6.00E-06	2	1.002	6	2.223	6	1.459	10.710
20000	3.71E+10	6.17E-06	2	1.003	6	2.318	11	2.521	24.340
30000	4.52E+10	6.31E-06	2	1.007	6	2.367	39	6.229	33.246
40000	5.82E+10	7.08E-06	2	1.021	7	2.669	137	59.332	47.190
50000	4.90E+10	8.71E-06	3	1.026	114	67.783	177	47.711	81.266
60000	3.71E+10	3.72E-05	2	1.017	158	116.040	194	21.107	93.106
70000	3.57E+10	3.80E-04	79	1.432	166	121.114	225	15.901	98.076
80000	2.68E+10	6.92E-03	201	111.297	4	2.399	16	2.405	93.109
90000	2.64E+10	4.57E-01	224	140.589	6	2.255	14	2.594	95.710
100000	2.44E+10	2.09E+01	253	165.974	9	2.223	9	1.821	95.808

Table 5.2. Multicast3 - Pattern 3 (Heavy Traffic)
Network Size: 8×8×8, 30% Multicasts

Graph 5.4. Unicast, Multicast1, Multicast2, and Multicast3 - Average Latency vs. Lambda (Pattern 3 - Heavy Traffic) 30% Multicasts on 4×4×4 Torus




Graph 5.5. Unicast, Multicast1, Multicast2, and Multicast3 -
Average Latency vs. Lambda (Pattern 3 - Heavy Traffic)
50% Multicasts on 4×4×4 Torus
1.00E-03
4.50E-04
1.00E-04
1.00E-05
1.00E-06
Average Latency (sec)
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
110000
120000
130000
140000
150000
160000
170000
180000
190000
Lambda (Packet/sec)
Unicast
Multicast1
Multicast2
Multicast3
CCITT Standard

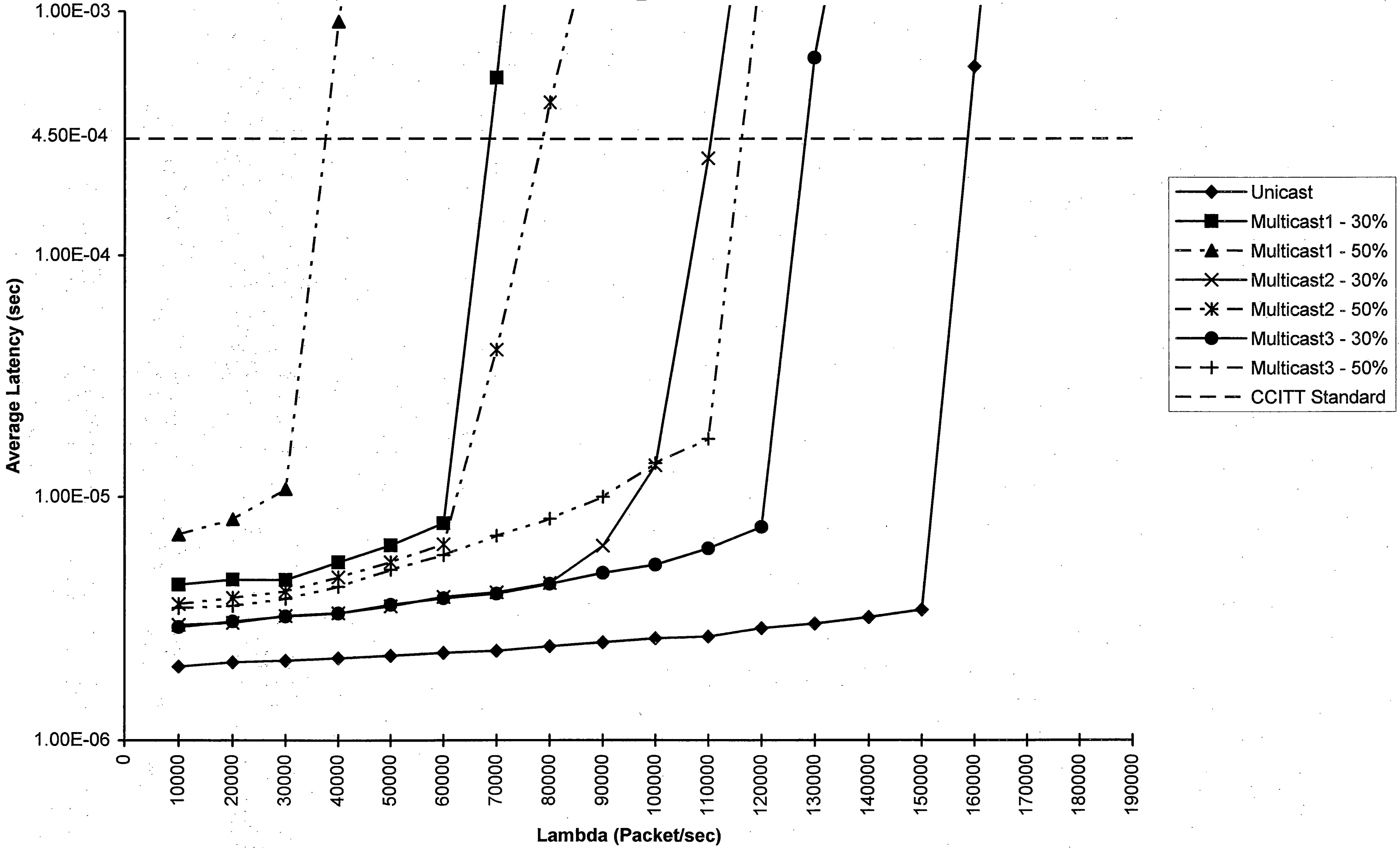
5.10 COMPARISON OF THE MULTICAST ALGORITHMS

Graph 5.6 shows the simulation results of 30% multicasts and 50% multicasts together for a network size of 4×4×4 and pattern 3 (heavy traffic). Unlike the latency curves of Multicast Algorithm 1 and Multicast Algorithm 2, the two latency curves of Multicast Algorithm 3 are very close to each other. This indicates that Multicast Algorithm 3 is much more sustainable than the other two multicast algorithms in the sense that it can handle higher traffic rates without degrading its performance. Also, Multicast Algorithm 3 with 50% multicasts performed better than Multicast Algorithm 2 with 30% multicasts.

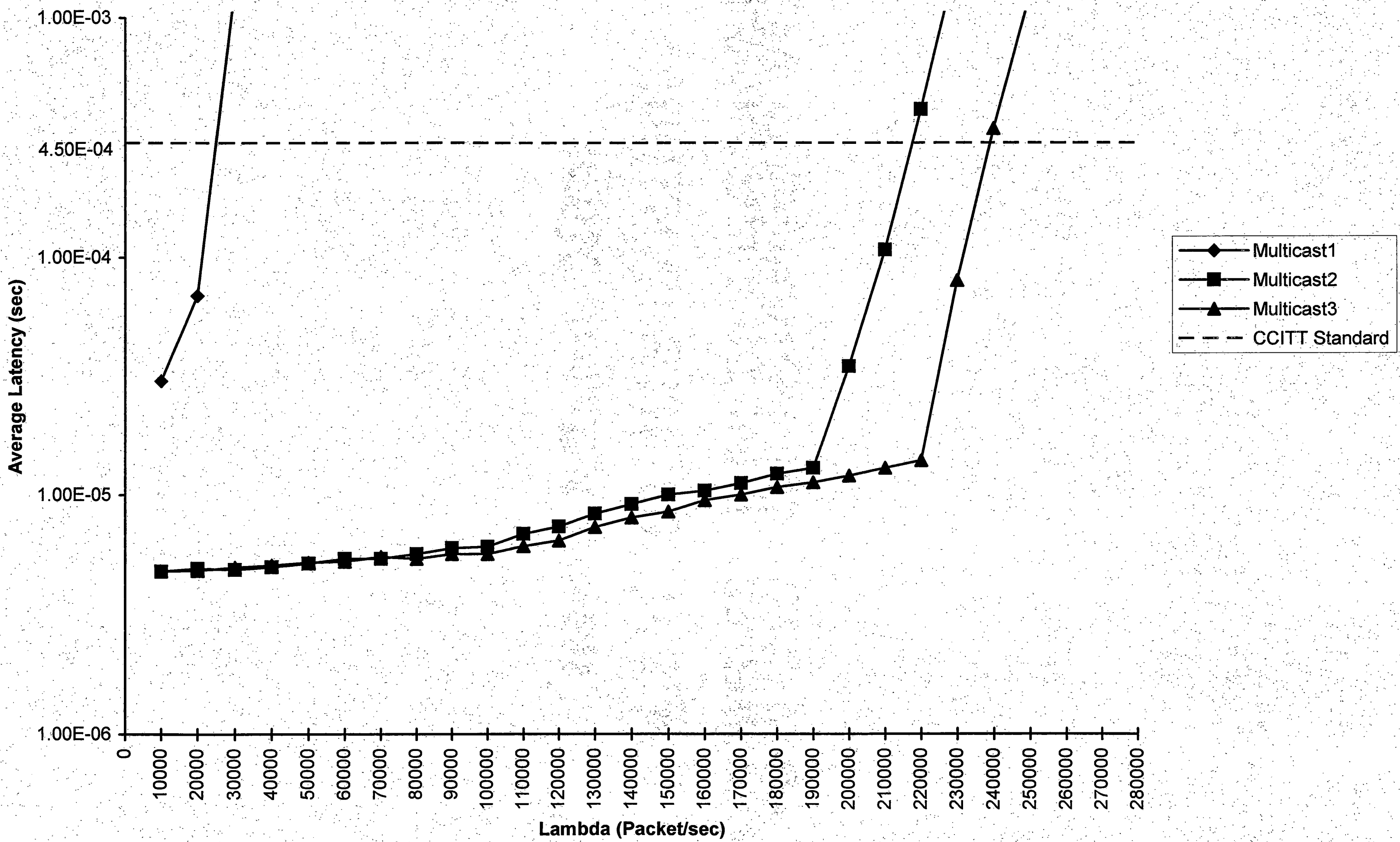
The simulation results of pattern 2 (medium traffic) came out to be the same except that the latency curves are shifted to the right. The simulation results of pattern 1 (moderate traffic) are not interesting since the latency curves are flat. However, Multicast Algorithm 1 shows an increase in latency time.

Graph 5.7 shows the result of a single source broadcasts. One selected node continuously issues broadcast packets. The performance difference between Multicast Algorithm 1 and Multicast Algorithm 2 is obvious. Multicast Algorithm 1 cannot support this simulation pattern at all. Similar to the other simulation results, Multicast Algorithm 3 performs the best among all.

Graph 5.6. Comparison of Multicast Algorithms (Pattern 3 - Heavy Traffic) on 4×4×4 Torus



Graph 5.7. Comparison of Multicast Algorithms for Single Source Broadcasts on 4×4×4 Torus



CHAPTER 6 -- EXTENSION AND CONCLUSION

In this chapter, several extensions to the multicast algorithms to improve their performance are discussed.

6.1 EXTENSION TO THE MULTICAST ALGORITHMS

The first extension to the multicast algorithms is to increase the size of each central queue so that they can hold more packets. The routing algorithms will remain the same. This will alleviate or postpone the congestion problem.

In this work, it has been assumed that communication channels are not multiplexed to keep the simplest form. To apply multicast algorithms to ATM switches, it is necessary to make better use of communication channels to increase network throughput. By time multiplexing each channel, a single physical channel can be thought of as multiple channels. This technique is called *virtual channels* [21]. It is possible to have a multiple set of central queues in each node by assigning a virtual channel to each set of central queues. In this method, each node can hold more packets and the communication channels will be highly utilized.

Extending the multicast algorithms to larger packets, it is possible to apply virtual cut-through as a routing method to hide latency. This enhancement is not suitable for ATM traffic.

6.2 FUTURE WORK

An integrated circuit design CAD tool, such as Magic, can be used to implement and test the base units. Also, the optimization of network throughput for the multicast algorithms needs to be studied as it applies to ATM switches. Ignoring the scalability, larger queue size for larger networks might decreases network latency even further. To find the correlation between queue size and network size, future research can be pursued by either simulations or probabilistic models. In addition, application of these algorithms to fault tolerant routing algorithms can be studied.

6.3 CONCLUSION

Two new multicast routing algorithms for torus networks of arbitrary size and dimension are presented. If a conventional unicast algorithm is used to handle multicasts, sudden increases in communication latencies are not avoidable (Multicast Algorithm 1). Multicast Algorithm 2 reduces the latency by using the same number of central queues as the unicast algorithm [5]. Multicast Algorithm 3 reduces the latency significantly by using separate queues for multicast operations. The torus network has significant advantages over the mesh. However, the presence of cycles in each dimension makes the development of routing algorithms on torus networks difficult. It is hoped that this work will contribute to the development of parallel computers and ATM switches using torus networks.

REFERENCES

[1] R. Y. Awdeh and H. T. Mouftah, "The Split-Switching Network (SSN): A High-performance Multicast ATM Switch," International Journal of Communication Systems, Vol. 8, pp. 191-202, 1995.

[2] D. Basak and D. K. Panda, "Scalable Architectures with k-ary n-cube cluster-c Organization," The proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing, 1993, pp. 780-787.

[3] L. N. Bhuyan and D. P. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Network," IEEE Transactions on Computers, Vol. C-33, No. 4, pp. 323-333, April 1984.

[4] R. V. Boppana and S. Chalasani, "Fault-Tolerant Wormhole Routing Algorithms for Mesh Networks," IEEE Transactions on Computers, Vol. 44, No. 7, pp. 848-864, July 1995.

[5] R. Cypher and L. Gravano, "Storage-Efficient, Deadlock-Free Packet Routing Algorithms for Torus Networks," IEEE Transactions on Computers, Vol. 43, No. 12, pp. 1376-1385, December 1994.

[6] W. J. Dally and C. L. Seitz, "The Torus Routing Chip," Distributed Computing, Vol. 1, No. 3, pp. 187-196, September 1986.

[7] W. J. Dally and C. L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," IEEE Transactions on Computers, Vol. C-36, No. 5, pp. 547-553, May 1987.

[8] M. De Prycker, *Asynchronous Transfer Mode*, Ellis Horwood, New York, NY, 1993.

[9] A. Fisher and H. T. Kung, "Synchronizing Large VSLI Processor Arrays," IEEE Transactions on Computers, Vol. C-34, No. 8, pp. 734-740, August 1985.

[10] D. Gelernter, "A DAG-Based Algorithm for Prevention for Store-and-forward Deadlock in Packet Networks, "IEEE Transactions on Computers, Vol. C-30, No. 10, pp. 709-715, October 1981.

[11] P. T. Gaughan and S. Yalamanchili, "Adaptive Routing Protocols for Hypercube Interconnection Networks," IEEE Computer, Vol. 26, No. 5, pp. 12-23, May 1993.

[12] C. J. Glass and L. M. Ni, "The Turn Model for Adaptive Routing," Journal of the Association for Computing Machinery, Vol. 41, No. 5, pp. 874-902, September 1994.

[13] L. Gravano et al., "Adaptive Deadlock- and Livelock-Free Routing With all Minimal Paths in Torus Networks," IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 12, pp. 1233-1251, December 1994.

[14] Q. Gu and J. Gu, "Two Packet Routing Algorithms on a Mesh-Connected Computer," IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 4, pp. 436-440, April 1995.

[15] K. D. Günther, "Prevention of deadlocks in packet-switched data transport systems," IEEE Transactions on Communications, Vol. 29, No. 4, pp. 512-524, April 1981.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Metro, CA, 1996.

[17] R. M. Hord, *Parallel Supercomputing in MIMD Architectures*, CRC Press, Inc., Boca Raton, FL, 1993.

[18] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc., New York, NY, 1984.

[19] K. Hwang, *Advanced Computer Architecture Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., New York, NY, 1993.

[20] A. Jajszczyk and W. Kabacinski, "A Growable ATM Switching Fabric Architecture," IEEE Transactions on Communications, Vol. 43, No.2/3/4, pp. 1155-1162, March/April 1995.

[21] P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Computer Communication Switching Technique," Computer Networks, Vol. 3, pp. 267-186, 1979.

[22] S. Lee, "Circuit-Switched Broadcasting in d-Dimensional Tori and meshes," The proceedings of Eighth International Parallel Processing Symposium, 1994, pp. 554-559.

[23] S. Lee and K. G. Shin, "Interleaved All-to-All Reliable Broadcast on Meshes and Hypercubes," IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 5, pp. 449-458, May 1994.

[24] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays Trees Hypercubes*, Morgan Kaufmann Publishers, Inc., San Metro, CA, 1992.

[25] P. M. Merlin and P. J. Schweitzer, "Deadlock Avoidance in Store-and-Forward Networks-I: Store-and-Forward Deadlock," IEEE Transactions on Communications, Vol. COM-28, No. 3, pp. 345-354, March 1980.

[26] P. Mohapatra and C. R. Das, "On Dependability Evaluation of Mesh-Connected Processors," IEEE Transactions on Computers, Vol. 44, No. 9, pp. 1073-1084, September 1995.

[27] J. Y. Ngai and S. Dhar, "A Deadlock-free Routing Control Algorithm for Torus Network based ATM Switches," The proceeding of SUPERCOMM/ICC, 1992, pp. 709-713.

[28] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, Inc., New York, NY, 1987.

[29] M. J. Quinn, *Parallel Computing Theory and Practice*, McGraw-Hill, Inc., New York, NY, 1994.

[30] R. Rooholamini and V. Cherkassky, "Finding the Right ATM Switch for the Market," IEEE Computer, Vol. 24, No. 4, pp. 16-28, April 1994.

[31] W. Stallings, *Data and Computer Communications*, Macmillan Publishing Company, New York, NY, 1994.

[32] Y. Tsai and P. Mckinly, "A Broadcast Algorithm for All-Port Wormhole-Routed Torus Networks," The Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation, 1995, pp529-536.