An Isotach Network Simulator

Technical Report CS-98-04

Lance S. Hopenwasser

Department of Computer Science, University of Virginia,
Charlottesville, VA 22903–2442
hopenwasser@acm.org
March 11, 1998

Abstract

We have developed a simulator of Isotach systems which run on a Myrinet network. The simulator allows us to study the performance characteristics of the system and provides a means for experimenting with alternative implementations of Isotach networks. The rate at which tokens flow through the network determines the rate at which Isotach logical time progresses; therefore, token behavior is of keen interest. The simulator gathers performance data that is used to examine the behavior of tokens within the network under several different conditions. The simulator is also capable of obtaining similar data on barriers that make use of the Isotach network. These barriers are of interest, because they can be used to establish network-wide checkpoints in addition to other application-level uses. The simulator can provide data on characteristics of the network that most affect barrier completion time. Furthermore, the simulator can be used to compare the Isotach barrier algorithm to a simple centralized non-Isotach barrier algorithm. We present some of the issues that influenced the design of the simulator.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Goals, Motivations, and Contributions

The main contribution of this project is a simulator that models an Isotach [14] system running on a Myrinet network [3]. The simulator is intended to be useful for studying the performance of Isotach networks as well as providing a means for experimenting with alternative implementations of Isotach networks. There are currently two prototypes of the Isotach system in development: a completely commercial off-the-shelf (COTS) version, in which the Isotach guarantees are implemented in software, and an implementation that makes use of custom hardware components as well as COTS hardware. For more information about the software implementation, see [13]. This simulator models the second implementation, which we discuss in more detail in Sections 2.2 and 2.3.

This simulator will be used for further studies on Isotach system behavior, including specifically some studies of the flex algorithm, an alternative algorithm for implementing Isotach systems. The simulator is designed to gather performance data for examining the behavior of tokens in the network under several different conditions. The rate at which tokens flow through the network determines the rate at which Isotach logical time progresses; therefore, token behavior is of keen interest. In addition, the simulator is capable of obtaining similar data on barriers that make use of the Isotach network. Barriers can be used to establish network-wide checkpoints in addition to other application-level uses. The simulator can provide data on characteristics of the network that most affect barrier completion time and can furthermore be used to compare the Isotach barrier algorithm to a simple centralized non-Isotach barrier algorithm. Finally, the simulator has been

designed to facilitate further development for unforeseen future experimental needs.

1.2 Overview

First we present the project background (Section 2) in which we give a brief description of Isotach systems, followed by more detail about those aspects of Isotach that are fundamental to the specific areas the simulator has been designed to address. Next, in Section 3 we describe the development of the Isotach simulator, including a brief discussion of previous work on earlier simulators that went into the development of the current simulator. This section includes an overview of the simulator design, followed by more detailed descriptions of selected implementation aspects. In the conclusion (Section 4), we discuss further applications of and some of the many possible enhancements to the simulator. In Appendix A we discuss how to use the simulator including how to use the parameter mechanism. In addition, we describe how to properly use the simulator to acquire statistically accurate data. Appendix B is a detailed and technical look at the simulator itself. This section should be useful for further development. Finally, a graphical representation of some of the predefined network topologies included in the code release is given in Appendix C.

Chapter 2

Background

2.1 What is Isotach Logical Time?

The essence of Isotach networks is to provide an inexpensive method for creating a total ordering in a distributed system [14, 15]. Isotach logical time is defined by an extension to the rules of logical time as described by Lamport in [9]. In general, logical times can be represented by lexicographically ordered n-tuples of integers. Isotach logical time is represented by a three-tuple in the format of (*pulse*, *pid*, *rank*) where the pulse is the basic unit of logical time. The pid is the process number of the source and is used to consistently order messages with the same pulse number. Finally, rank is the order in which messages were sent from a given pid. Thus, every message is given a unique logical time. Isotach's extension to logical time is called the *Isotach invariant*. This extension is the guarantee made by the Isotach network that a message sent at time (i, j, k) will be received at time ($i + \delta$, j, k), where δ is defined as the logical distance between the sender and receiver. In other words, a message travels one unit of distance in exactly one pulse. The prediction of the logical receipt time of a message given a particular send time enables logical synchronization of asynchronous processes at low cost.

For simplicity, in the prototype, the logical distance between two hosts is defined as the number of switches traversed by a message. The network diameter **d** is the largest logical distance present in the network and is thus the longest amount of logical time any Isotach message can take to travel between a sender and a receiver. The pid is picked arbitrarily (but consistently), and the rank is provided implicitly by the FIFO nature of the Myrinet network upon which the Isotach Network is built. The actual progression of logical time in the network is achieved through the use

of special messages called tokens. From the point of view of the senders and receivers, the receipt of a token from each neighbor indicates the end of one pulse and the start of the next pulse. The actual mechanism for the transmission of these "waves" of tokens is described in the next section.

2.2 The Token Mechanism in an Isotach Network

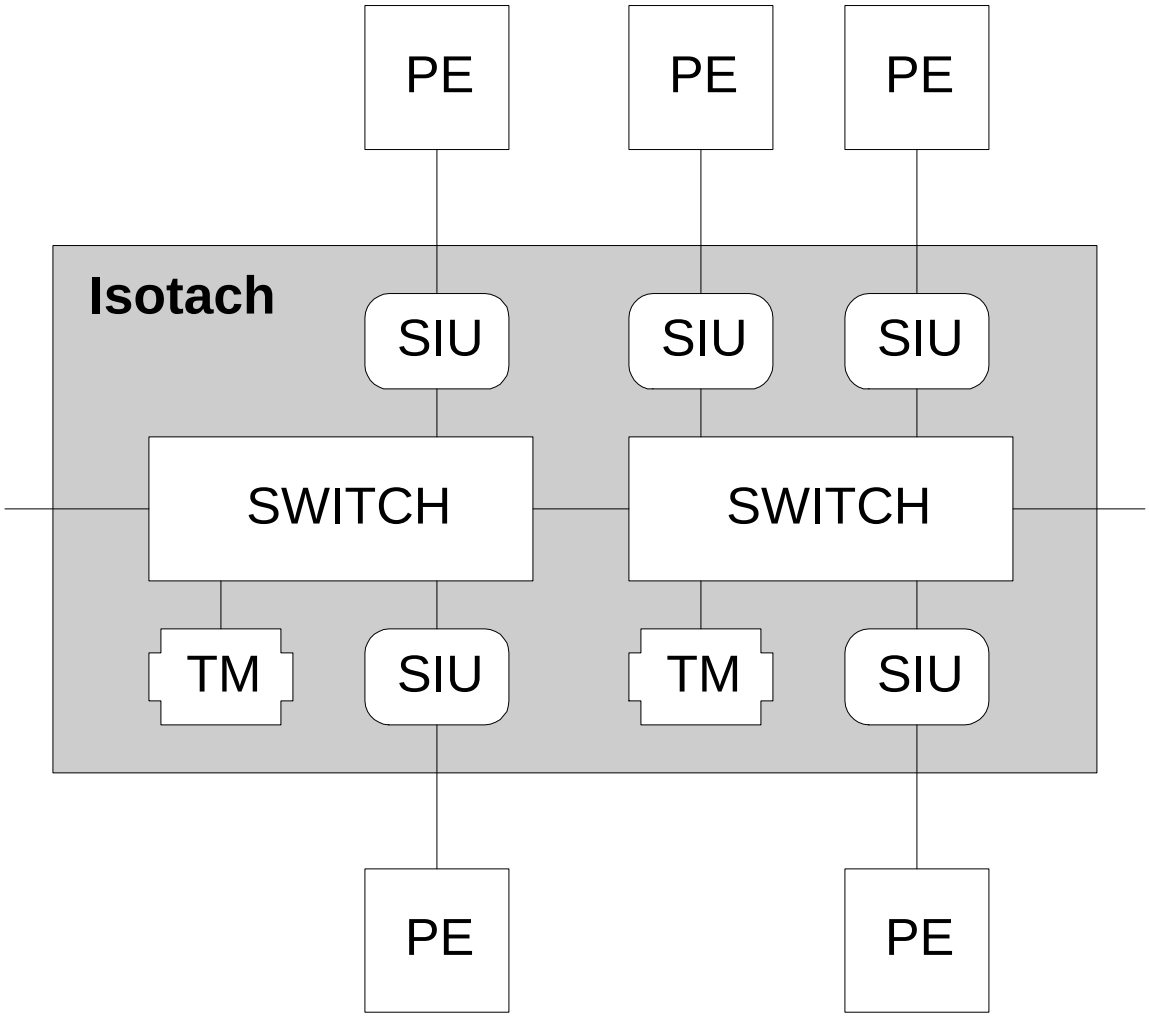


Figure 2.1: **A sample Isotach network**

Each switch is connected to one TM and some number of other switches and PE/SIU pairs. The switches, TMs, and SIUs maintain Isotach time, whereas the PEs are not directly involved with the Isotach specifics.

During normal operation of the prototype, the hosts, or Processing Elements (PEs), are not actually involved with the transmission of tokens. The waves of tokens are created by Switch Interface Units

(SIUs) and Token Managers (TMs). The SIUs sit in-link between a PE and its switch, and each switch has one TM attached to it. An example configuration is shown in Figure 2.1. Since tokens are only sent between SIUs and TMs, Isotach logical time is contained within the perimeter of the SIUs; PEs do not directly participate in Isotach logical time.

Figure 2.2 shows a *time diagram* of the actual token mechanism within a small Isotach network. This diagram represents the token traffic in a small network consisting of two SIUs and two TMs. Time progresses in the direction of the arrows, and the lines connecting the timelines represent the tokens travelling in the network. An SIU sends tokens to and receives tokens from its neighboring TM only (i. e., the TM attached to the same switch as the SIU). The SIU simply sends an initial token and then sends a token each time it receives a token. A TM sends tokens to and receives tokens from all its neighboring SIUs and all its neighboring TMs (i. e., the TMs attached to the switches adjacent to the sending TM's switch). A TM initially sends a token to each neighbor. After it receives the i^{th} token from each neighbor, it sends out the $i+1^{\text{st}}$ wave of tokens and waits again for the $i+1^{\text{st}}$ token from each neighbor.

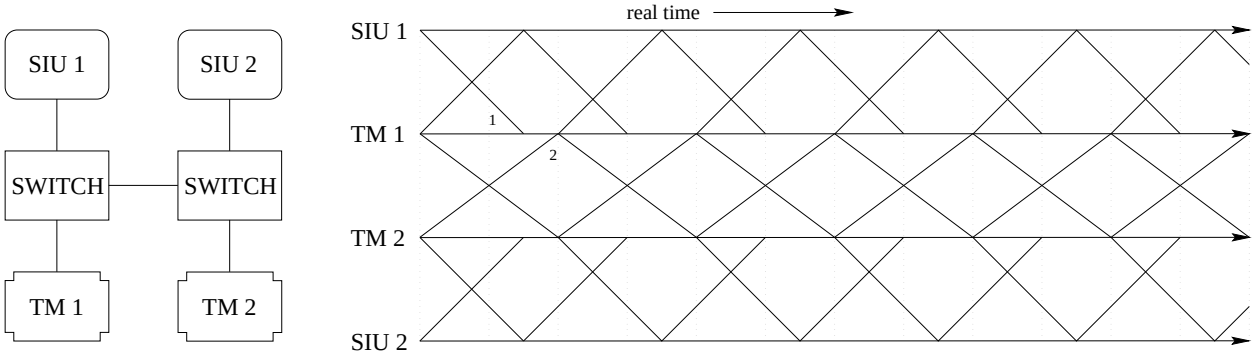


Figure 2.2: **The token mechanism in a small Isotach network**

The SIUs send a token for each token received. The TMs send a wave of tokens when tokens have been received from all neighbors. For example, TM 1 first receives a token from its neighbor SIU at point 1. When it receives a token from its neighbor TM at point 2, it has received a token from all its neighbors, so it sends out the next token wave.

2.3 Barriers in an Isotach Network

As stated earlier, barriers can provide a mechanism for establishing checkpoints across the entire Isotach network. In addition, many applications make use of barriers, and the performance of

barrier completion is often of critical importance. Within the Isotach network, barriers are piggy-backed on tokens for efficiency; we refer to these as barrier tokens. In the prototype, barrier tokens are simply normal tokens with a barrier indication bit set. For the TMs, the barrier algorithm is very similar to the regular token algorithm. Initially, every TM sends a barrier token to all of its neighbors. After this initial wave, a TM sends out the next wave of barriers when it has received a barrier token from each of its neighbors. A PE, when it reaches its barrier locally, informs its SIU by sending a message called a barrier marker. Once a PE has sent a barrier marker, it does not send another until it receives a barrier completion notification from its SIU indicating that all other PEs in the network have reached their barriers. In other words, a PE only participates in one barrier at a time. When an SIU has received both the PE's barrier marker and a barrier token from its TM, it proceeds to send a barrier back to its TM. Note that if the SIU receives a barrier token from its TM but has not received its PE's barrier marker, it just "remembers" that is has received a barrier token and returns an "ordinary" token to its TM (i. e., a token without the barrier indication bit set). When the SIU receives a barrier marker from its PE, it can send the next outgoing token as a barrier token. After it has sent back its first barrier token, the SIU sends a barrier token for each barrier token it receives. When the SIU has received $\mathbf{d}+1$ such tokens, it knows that every PE has reached its barrier, so it can send a barrier completion notification to its PE. To allow for the reuse of barriers, for each barrier wave, an SIU stops sending barriers after it has received and sent exactly $\mathbf{d}+1$ tokens.

To demonstrate the correctness of this algorithm one must show that two properties hold: first, no PE will be notified of barrier completion before all PEs have reached their barriers, and second all PEs will be notified of barrier completion once all PEs have reached their barriers. We can informally prove that the first property holds by looking at the case where one PE does not send a barrier marker. If PE_i does not send a barrier marker to SIU_i, the SIU for PE_i, then SIU_i will not send a barrier token to TM_j, SIU_i's TM. Without this barrier token, TM_j can send at most one wave of barrier tokens. Thus any other SIUs connected to TM_j (i. e., at distance 1 from SIU_i) can receive at most one barrier token. With only one barrier token from TM_j, a neighboring TM at distance one from TM_j, can send at most two waves of barrier tokens. Thus the SIUs connected to this neighboring TM are at distance two from SIU_i and can receive at most two barrier tokens. Similarly, since each additional unit of distance from SIU_i adds one to the maximum number of tokens that can be received, a simple inductive argument shows that an SIU at any distance $\boldsymbol{t}$ from SIU_i can receive at most $\boldsymbol{t}$ barrier tokens. Therefore, in an Isotach network of diameter $\mathbf{d}$, an SIU

that is the farthest possible distance from an SIU that has not reached its barrier, can receive at most **d** barrier tokens. Thus no SIU can receive the $\mathbf{d}+1^{\text{st}}$ token required to notify its PE of barrier completion until all PEs have reached their barriers.

Since every host is notified of barrier completion when its SIU receives the $\mathbf{d}+1^{\text{st}}$ token, we can informally demonstrate that the second property holds by showing that if every host reaches its barrier, every SIU must receive its $\mathbf{d}+1^{\text{st}}$ barrier token. First assume that all hosts have reached their barriers, but that some SIU_a fails to receive its $\mathbf{d}+1^{\text{st}}$ barrier token. Since every TM emits barrier token wave $i+1$ upon receiving barrier token wave i, we know that the TMs can not be responsible for holding back the progression of barrier tokens through the system. Thus if the next expected barrier token is not received at SIU_a, it must be because one or more other SIUs have failed to return a barrier token. If SIU_a does not receive barrier token $\mathbf{d}+1$, some other SIU must have failed to return barrier token k, where $k < \text{d}+1$. A failure by an SIU to return a barrier token in a barrier token wave later than **d** has no affect on whether SIU_a will receive barrier token $\text{d}+1$. However, every SIU unconditionally returns d barrier tokens after its host reaches its barrier. Therefore, SIU_a must receive barrier token $\text{d}+1$.

Chapter 3

Development of the Isotach Simulator

3.1 Previous Work and Overview of the Simulator Design

The Isotach network simulator is time-stepped and consists of two major parts: the network layer and the Isotach layer. The network layer part of the simulator is based on a simulator written by the Avalanche group [1] which models messages as they traverse the Myrinet network. See Appendix B for details concerning the actual structure and legacies of the original Myrinet simulator. The network, which uses wormhole routing, was originally modeled by a configurable topology of hosts, switches, and links. The links represent the wires connecting nodes and switches. The Avalanche group's original simulator was first modified by Frank Brill. These modifications consisted of streamlining the Myrinet simulator by removing a large portion of the underlying Mint simulation system. We further modified this simulator by removing the remaining Mint code. In addition, we re-designed the switch mechanism (see Section 3.3) and parameterized the latencies associated with the switches and links. This allows each link in the simulator to be specified by one of two different link latencies representing the "near" and "far" link types in a real Myrinet network.

In addition to the modifications and redesign of the Myrinet layer, we designed an Isotach layer which we integrated with the Myrinet layer. The Isotach layer models the Isotach hardware: the TMs, the SIUs, and in a very limited sense, the PEs. The TMs are simple elements that collect and send tokens. The simulator models the core functionality of the TMs in the prototype, but lacks the fault-tolerance and Isotach signal mechanisms. In the prototype, the SIUs are composed of smaller "machines" connected by FIFO buffers. These machines work in parallel to process all Isotach traffic. The simulator does not yet model the complete functionality of the SIUs. Like the TMs,

the SIUs perform the core token handling functionality without the fault-tolerance algorithms and the Isotach signals. Further, it only minimally models the SIUs' operations on PE to PE Isotach traffic, in that an SIU timestamps outgoing Isotach messages so that they are, in a limited sense, constrained by the rules of logical time, but does little else. Isotach messages that are received by the destination PE's SIU undergo no processing; they are simply forwarded to the PE. Non-Isotach traffic correctly bypasses the SIUs' internals; it is not buffered at each SIU. Finally, the PEs are simply generators of non-Isotach and Isotach traffic (other than tokens). The Isotach traffic can include barrier notifications to the SIU, and the distribution of the rest of the messages sent depends upon the experiment and type of generated traffic.

The passage of time in a simulation is measured in abstract units of time called cycles. A cycle is defined as the time it takes to move one flit within the network. In the current Myrinet release, a link can process one flit each 6.25 ns. Therefore, we can map one cycle of simulation time to 6.25 ns of wall-clock time. The speed, in cycles, of each of the Isotach elements is parameterized separately, and the speed of each of the machines within the SIU is parameterized relative to its SIU's speed. In both our simulator and the prototype, all the Myrinet links between SIUs, TMs and switches are near links and all the Myrinet links between PEs and SIUs are far links. Additionally, all of the switches are modeled as having the same fall through delay (near to near). Given the current Myrinet network, the near links have a latency of approximately 28ns and the far links connecting PEs and SIUs have a latency of about 110ns, thus these links were modeled as 4 flits long and 18 flits long respectively. The fall through delay of a switch connecting two near links is approximately 100ns, so the simulator's switch fall through delay was set to 16 cycles. Finally, according to hardware specifications tokens are 6 bytes, so tokens are 6 flits plus one *routing flit* per switch along the token's route.

3.2 Mechanism for Moving Flits

The simulator provides two components that make up the fabric of the network: 1) buffers and 2) links. A network is composed of alternating buffers and links. Buffers simply provide space for flit storage along the message's path, although it is important to note that it takes at least one cycle to cross the buffer. The links, through the use of a bitstream, model the latency from one node to another.

For historical reasons, information about the locations of flits within the network is kept on a per message basis. Therefore each cycle the flits of one message are moved independent of the flits of the other messages. The message's flits are moved by starting with the buffer or link occupied by the head flit of the message, following the interconnected buffers and links backwards along the message path, and ending at the buffer or link occupied by the tail flit. Each buffer or link is responsible for moving a flit from its upstream connection (if there is a connection and a flit is available) into itself. Since links are represented by bitstreams (see Appendix B), this behavior means that in each cycle a link is only concerned with right-shifting its contents and placing either a 0 or a 1 at its leftmost position depending upon whether or not an upstream flit is available. Whether the flit lost off the right-hand side is a 0 or a 1 is of concern only to the downstream buffer, which should have already been taken care of for the current cycle. On the other hand, it is the link's duty to decrement its upstream buffer's flit count if the link receives a flit from upstream. Buffers merely maintain a flit count, so it simply increments the count if it receives a flit from its upstream link. It does not actually need to do anything to the link, because the link will right-shift its contents anyway.

Flow control is managed through the use of STOP and GO signals sent by a buffer to its upstream link. When a buffer fills up past its parametrically defined high watermark, it sends a STOP signal upstream, which is received after a propagation delay (defined through parameters). Any flits that are currently in this upstream link after it has been STOPped proceed through the rest of the link and empty into the downstream buffer (the one that sent the STOP signal). However, the STOPped link does not retrieve a flit from its upstream buffer.[1] Similarly, when a buffer's flit count drops below its low watermark, it sends a GO signal to its upstream link (which is also received after the propagation delay), signalling the upstream link that it can now retrieve a flit from its upstream buffer.

3.3 Re-designing the Simulator's Myrinet Switch

The switch is essentially implemented as an array of buffers and links, where each array index corresponds to a switch port number. When a message arrives at a buffer (the switch's input port), the downstream link (output port) of that buffer is dynamically chosen based upon the message's

[1] Thus this buffer can now fill up, send a STOP signal to its upstream link, and so on. In this way, the flow control can propagate all the way back to the sending node.

routing information. In the original simulator, the time it took a message to traverse a switch (fall through delay) was implemented using a simple counter. When the head flit arrived at a buffer, the counter was started. Each cycle the counter was incremented until it reached the appropriate delay value, at which time the downstream link was connected to the input port's buffer and the message was allowed to stream through at no additional cost. The routing flit was also dropped at this time. A diagram of the original switch design is shown in Figure 3.1.

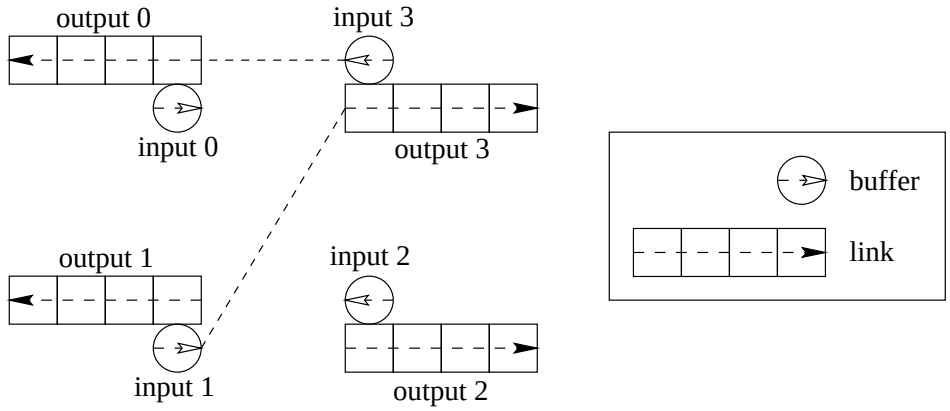


Figure 3.1: **The original switch design**

The original switch design consisted solely of input and output ports, where the input ports were buffers and the output ports included the link to the next node. The crossbar latency was modeled using a simple counter, and once the counter finished a connection was made from an input port to an output port. In this diagram, input 1 is connected to output 3 and input 3 is connected to output 0.

Through performing several experiments, we discovered that the order in which a TM sent out a wave of tokens affected the average token pulse length. We traced this behavior to the fact that many tokens arrived back to back at the input port of the TM's switch. Later tokens were being penalized part or all the cost of the previous messages' fall through delay in addition to the time it took previous messages to vacate the input port. This behavior is incorrect, because in reality, the switch is heavily pipelined. Although there is a small minimum delay between sequential messages due to the time it takes a message to traverse the input port, the crossbar is pipelined. Thus, later messages were being delayed significantly more than they should have been.

In order to fix this problem with the switch, we needed to model the switch more accurately. We increased the accuracy by replacing the fall through delay counter with a more sophisticated mechanism which involved modeling the crossbar and output port buffer as well as the already

implemented input port buffer. We achieved the pipelining behavior needed for the switch latency by making the crossbar a bitstream. In order to enable multiple messages coming from the same input port but going to different output ports to simultaneously traverse the switch, the "switching" is performed between the input port buffer and the crossbar.[2] There is one crossbar exclusively associated with each output port. Since the input buffer/crossbar connection cannot be broken while a message has some flits in both the input buffer and the crossbar, a message from a different input port but destined for the same output port is blocked. Once the tail flit of the blocking message enters the crossbar, the crossbar is free to be attached to another input port, so the blocked message can then enter the crossbar link. This is indeed a simplification of the real structure of a switch, where messages are literally blocked at cross-points within the crossbar grid. But given the heavy nature of the pipelining through a switch, we believe that this should not significantly affect the average timing.

Furthermore, the flow control mechanism within the switch is different from the previously described flow control. The switch output port has a buffer for flits coming from the crossbar. When the buffer reaches maximum capacity, the flits will start to back up within the crossbar itself. When the crossbar is backed up all the way to the input port buffer, then the input port buffer starts to fill up and behaves normally with regard to flow control.

The implementation of these changes in the simulator required a number of changes to the data structures at the Myrinet level. In order to reuse existing components, we decided to model the crossbars in the switch by using an additional array of link/buffer pairs. The link portion represents the crossbar latency, and the buffer portion represents the internal flit storage between the crossbar and the downstream inter-node link. A diagram of the new switch design is shown in Figure 3.2. As the diagram shows, each additional link/buffer pair, since it represents a crossbar, is associated exclusively with one inter-node link.

In particular, implementing this functionality required changes to both the buffer and link objects. First, the crossbar buffer does not use high and low watermarks; rather, it has a maximum capacity. Once this capacity is reached, it simply does not accept any more flits from upstream (the crossbar link). As described in Section 3.2, normally a buffer must accept a flit from its upstream link if it is available. Thus some additional logic needed to be added to the buffer object so that it

[2]Since we do not need the routing flit past the point of the actual switching, this flit (the head flit) is dropped before the message enters the crossbar.

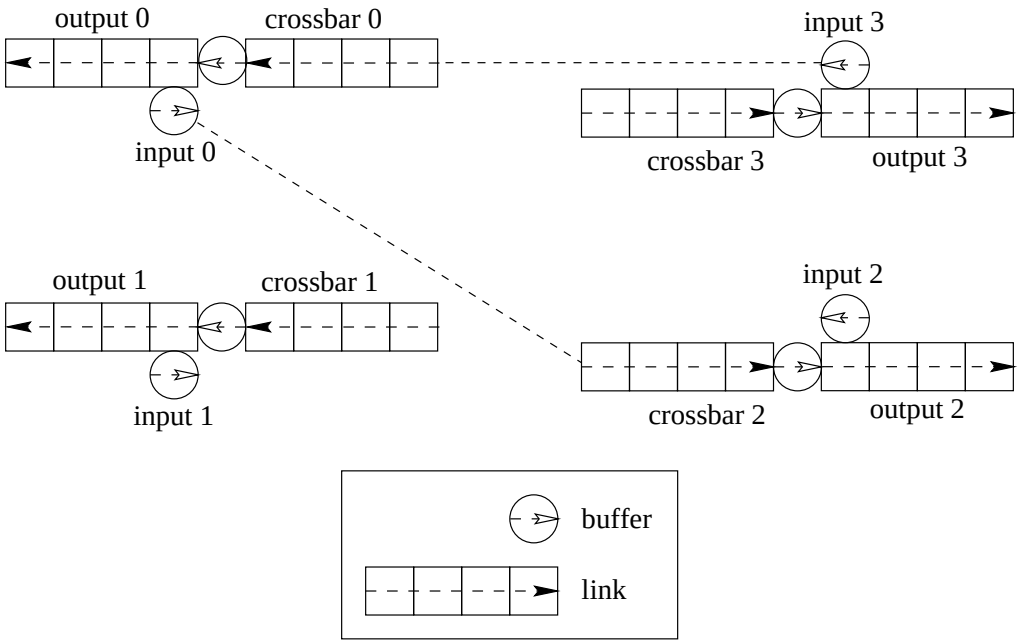


Figure 3.2: **The new switch design**

The new switch design includes an additional internal link and buffer to model the crossbar. Each crossbar is associated with a single output port, and connections are made between the input port buffers and the crossbar links. In this diagram, input 3 is connected to crossbar 0 and input 0 is connected to crossbar 2.

could refuse the incoming flit. Additionally, the buffer must notify the crossbar link that it did not accept the flit. It does this by sending a STOP signal to the link with no propagation delay. In other words, the crossbar buffer tells its upstream link that it does not want to receive any more flits, effective immediately.

However, with the crossbar buffer refusing the flit, the crossbar link, when STOPped, can no longer just right-shift its bitstream without regard to what happens to the rightmost bit. To further complicate matters, unlike the "regular" link, a STOPped crossbar link may or may not accept flits from its upstream buffer depending upon whether it is filled up or not. In other words, the crossbar link can effectively act as a buffer and store flits. However, in the original simulator, the links could not provide static storage of flits. As described above, once a flit entered the link it would proceed through the link every cycle (to the destination buffer). This is correct behavior for an inter-node link, but not for a crossbar link. When a flit is stopped in the crossbar (starting with a 1 in the rightmost position of the bitstream), all immediately following flits stop moving forward. Thus we also had to add some functionality to the link structure to allow for a link to be stopped

at its downstream end. Note that a flit within a stopped crossbar continues proceeding forward until it is blocked by another flit or has reached the rightmost end of the bitstream.

Since the link occupation data is actually stored on a per message basis, a message knows only about where its own flits are in the crossbar. If more than one message is in the crossbar, the second and greater message must have some means of accesing the previous message, so that it can retrieve the previous message's flit occupancy data. This data lets the message determine whether there is room for its head flit to proceed or not. This functionality also introduces a dependency of the movement of one message upon the movement of another. For this reason, some additional logic had to be added that both allowed a message's movement to be calculated "out-of-turn"[3] and that made sure the message's movement was only calculated once for any given cycle. Finally, if the link fills up, it cannot accept any more flits from the upstream buffer, so it is only at this point that it stops decrementing the flit count of its upstream buffer.

Other small, miscellaneous changes included parameterizing the buffer sizes, the buffers' high and low watermarks, and the propagation delay for signals sent across inter-node links. These values were originally defined as constants that were hard-coded, which restricted necessary flexibility. Additionally, we added a framework that supports two different sets of these values, one for near links and one for far links.

3.4 Known Inaccuracies in the Simulator

3.4.1 Switch Input Port Delays

Although the crossbar bitstream allows pipelining through the switch, there is in reality a minimum separation caused by some processing that the switch input port must perform on each message separately. This minimum seperation is, according to personal correspondence with the switch designer, approximately 20ns. The only time that this minimum delay has a noticeable affect is when two messages are back to back; the head flit of the second message is ready to go through the crossbar "immediately" after the tail flit of the first message has vacated the input port. Obviously, if the second message's head flit is more than 20ns behind the first message's tail flit, this minimum delay problem becomes irrelevant. If the head flit is less than 20ns behind the tail flit, but not 0ns behind, it is only penalized part of the delay. Thus the imposition of the whole delay upon the

[3] Normally, the network object simply goes through its list of current messages sequentially.

second message occurs almost only when the second message is already waiting in the switch input port. In practice, this penalty would not be incurred very frequently with respect to the number of messages that traverse the switches. For this reason, we deemed that the actual affect on the system would be minimal; therefore, the potential increase in accuracy is not worth the increase in overhead the implementation would cause.

3.4.2 SIU Bypass Latency

In the hardware implementation of the SIU, some messages (such as non-Isotach messges) are not consumed by the SIU; rather, they bypass the SIU while maintaining the streaming that occurs in the rest of the Myrinet network. This bypass naturally has some latency associated with it. The current simulator models this latency using pre-existing components, namely the links. As there are only two types of links, near and far, the bypass latency is constrained to one of these two values. We selected the near link for our implementation, since the near latency is likely to be much closer than the far latency to whatever the real bypass latency may be. Arguably, a third type of link (say *bypass*) should be created; however, this would require a significant amount of work. We do not yet have any reliable estimates of how long the bypass latency really is in the hardware. Since we need to make our best guess anyway, it is not unreasonable to set the bypass latency equal to the near latency and postpone the necessary extra code changes. This problem only affects traffic that must bypass the SIU; Isotach traffic and token traffic are not directly affected.

3.5 Correctness of Execution

3.5.1 Checking the Results

To check that the simulator model matches our intended design, we verify that the results of some simple runs match our predictions made with independent analysis. The first of the primary statistics used for this purpose are the average message latencies broken down by sender and receiver types. Three categories are based on the sender and receiver types: SIU→TM, TM→SIU, and TM→TM. The second set of statistics consist of measurements of the total amount of time a message is blocked. This second set of data is also averaged across messages broken down by sender and receiver types. Finally, to get statistically accurate results, we accumulate our data in accordance with our recommendations detailed in Appendix A.6.

Through some simple independent analysis we can determine what the minimum latency for each category is. This minimum latency is equivalent to the latency of the message without any blocking. To determine the minimum latency, we must examine the path from source to destination. Each link on this path (both inter-node and intra-switch) takes as many cycles to traverse as the defined length of that link. The links between TMs and switches and between SIUs and switches are defined as near links and are 4 cycles long. The crossbar link is defined to be 16 cycles long. According to the simulator design, each buffer (excluding the final destination's buffer) along the messge's path takes a minimum of 1 cycle to cross. The Myrinet network uses source routing, meaning that the first flit of a message entering a switch is consumed to determine the exit port. Since the head flit is consumed, the measurements of latency across the switch must take into account that the time is measured from when the head flit enters the switch to when the second flit (now the head) exits. This means that each switch traversed adds one cycle to the total message latency. The destination must wait for each flit in the message to arrive at the cost of one cycle per flit. Finally, the destination takes one additional cycle to consume the message once all flits have arrived at its input port.

Determining the minimum latency then becomes a simple case of addition. The paths of both SIU→TM and TM→SIU consist of two near links, one crossbar, one switch, and two buffers. In our "verification" simulation, we use exclusively token traffic, thus the size of the message arriving at its destination is six flits. Finally, we must add one cycle for the destination to consume the message. Adding all of this together we get:

$$\underset{(2*4)}{\text{links}} + \underset{16}{\text{crossbar}} + \underset{1}{\text{head flit}} + \underset{(2*1)}{\text{buffers}} + \underset{6+1}{\text{destination}} = 34$$

For the TM→TM path, we have three near links, two crossbars, two switches, four buffers, and the destination. Thus the minimum latency for TM→TM is:

$$\underset{(3*4)}{\text{links}} + \underset{(2*16)}{\text{crossbars}} + \underset{(2*1)}{\text{head flits}} + \underset{(4*1)}{\text{buffers}} + \underset{6+1}{\text{destination}} = 57$$

Since the minimum latency is equivalent to the actual message latency minus the amount of time spent blocked, we can check the results by subtracting the measured average blocked cycles from the measured average total message latency and verifying that the result is equal to the calculated minimum mesage latency. Sample results from one simulation are shown in Table 3.1. As can be seen from this table, the measured results from the simulation are very close to the calculated values in all categories.

		measured cycles					calculated cycles
		latency	–	**blocked time**	=	**unblocked latency**	**minimum latency**
TM→TM	μ	57.784	–	1.025	=	56.759	57
	σ	1.585		1.965		3.550	
TM→SIU	μ	34.000	–	0.000	=	34.000	34
	σ	0.000		0.000		0.000	
SIU→TM	μ	34.035	–	0.059	=	33.976	34
	σ	0.245		0.388		0.633	

Table 3.1: **Average latencies grouped by sender and receiver type**

The measured average latencies of each category of message minus the average blocked time is very close to the independently calculated minimum latency. These measurements were taken from the 3rb1 network topology (see Appendix C).

3.5.2 Internal Checking

We included several sanity checks within the program itself to ensure that the simulation is executed correctly. These checks are conducted during the course of a simulation. For example, to guarantee that no messages are lost (or added) at the network level, we keep a tally of the number of messages sent and received. At the end of a run, we verify that the number of messages received plus the number of messages remaining in the network (undelivered) is equal to the total number of messages sent. This information can be printed to a trace file for manual verification. As another example, we ensure that the Isotach invariant holds by checking that the timestamp of every Isotach message received at its destination SIU is not less than the SIU's current receive clock, because this relationship would indicate that the message is arriving late.

The simulator includes two internal tests to check that certain properties of barrier completion hold true[4]. The first property involves the number of barriers sent and received; a PE should not receive notification of the the completion of the i^{th} barrier unless all the PEs have reached i barriers. We ensure this property by keeping track of the number of barriers completed and reached at each PE. The second property requires that all messages sent within a given epoch be received at their destinations at the time of that epoch's barrier completion. We ensure this property by

[4]The barrier checks can impact the execution time, so they are enabled/disabled through the use of a parameter (`verifyBarrierCompletion`).

keeping track of the epoch in which each message has been sent. When a message arrives at its destination, we verify that its epoch number is not less than the number of barrier completions at its destination. We ran several simulations with both Isotach and non-Isotach barriers, and these checks indicate that the barrier algorithms are working correctly.

Furthermore, numerous assertion statements are used throughout the program. The user has the choice of compiling either with or without these statements. The tradeoff for including the statements is a loss of execution speed in favor of yet more reassurances that the program is executing correctly.

Finally, all random numbers used in the simulator are generated using LEDA (Library of Efficient Datatypes and Algorithms). The validity of the LEDA random number generators is discussed in [10].

Chapter 4

Conclusion

In conclusion, we developed a simulator that usefully models Isotach systems. This development involved understanding and modifying inherited code from a previous simulation. Since the previous simulation only modeled a part of the entire Isotach system, namely the Myrinet level, we performed a significant amount of object-oriented design and implementation to produce a full simulator that included both a Myrinet and an Isotach level. Furthermore, we fixed a serious problem in how the inherited code modeled the Myrinet switch. Finally, we developed a powerful mechanism for handling the large number of varying orders and types of parameters required by the simulator. This parameter mechanism, in conjunction with the heirarchy of classes used in statistical data gathering, provides a means of readily adapting the simulator to study other aspects of Isotach systems.

4.1 The Application and Usefulness of the Simulator

The simulator in its current implementation can be used for several different kinds of studies. For example, it would be useful to see how different configurations affect token turnaround time (which is effectively the length of the logical pulses). In addition to the ability to measure the various message latencies as shown above, the simulator includes facilities for measuring the average pulse lengths (as well as the standard deviations) in a wide variety of categories. For example, one can examine the pulse lengths at each TM or SIU separately or across the entire network.

The simulator can also be used to perform studies on barriers within an Isotach system. Facilities are included for both the Isotach barrier algorithm and a simple centralized non-Isotach algorithm.

Thus the simulator can be used to compare the effects of different topologies and configurations upon barrier completion time for each barrier algorithm.

Finally, the simulator is designed to enable studies that had not been developed during the design and main implementation of the simulator. For example, studies of the flex algorithm, an alternative algorithm for implementing Isotach systems, are scheduled to be performed in the near future. These studies will probably require some additional development, and the simulator is designed to provide a framework for this type of enhancement.

4.2 Future Work

A number of additions could be made to the simulator for examining other aspects of Isotach networks. The most obvious is the completion of the SIU and PE modules. Currently several of the SIU's internal "machines" are effectively only stubs, thus only minimal Isotach application level messages are supported. Completing the SIU and PE modules is needed for any studies about the Isotach network from an application's perspective. Other tasks include developing the SIU and TM classes to include support for tolerating lost tokens and for sending broadcast signals. These facets of the Isotach system are discussed in the "Design of the Isotach Prototype" working paper.

There are also several additions/modifications that would improve the usability of the simulator. Most notably, the form of the output of results is quite limited and should be expanded. In the current implementation, one must determine which statistics should be printed prior to the execution of the simulation. Once the simulation has finished, only these statistics are printed and all other data are lost as the program ends. This means that the simulation must be rerun if the statistics of interest change, even if all the parameters for the new simulation are identical to the parameters of the original simulation. A very useful improvement would consist of separating the statistics from the data. The simulation would print all data acquired through the run to a data file. This data file could then be read by a separate program that would print out only the pertinent statistics.

Appendix A

User's Guide

A.1 Compiling the Simulator

A make file for the simulator is provided in the release directory. This file provides four pre-configured compiling options: `debug`, `final`, `assert`, and `tuning`. The `final` option is selected by default. The `debug` option defines the `DEBUG` and `DEBUG_RND` flags as well as using the `-g` compiler flag to gather symbol data. The `DEBUG` flag enables some time-consuming sanity checks and the `DEBUG_RND` flag turns on repeatable random number generation (by setting the seed to a defined constant). This option is useful to the programmer who wishes to do further development on the simulator code. The `final` option defines the `NDEBUG` flag, which disables the `assert` statements, and it also uses the compiler flag `-fast` for maximum optimization in favor of speed.[1] This is the option that is recommended for performing actual simulations. The `assert` option is the same as the `final` option, except that it does not disable the `assert` statements. This option gives a little more security of correct execution, but does sacrifice some runtime speed. Finally, the `tuning` option is identical to the `final` option except that it also uses the `-g0` flag to obtain symbol information without turning off inlining. This option is useful for performance tuning and is also intended for the developer.

The simulator requires the LEDA libraries. By default the static libraries are used, because there is some indication that it gives a small gain in speed over dynamic linking (at the sacrifice of executable size). With this option, the actual libraries are necessary only at compile time. By

[1] The compiler flag `-fast` is used when the default compiler, Sun's CC, is selected. If g++ is the selected compiler, the compiler flag `-O` is used instead.

changing which definition is used for the `LIBS` makefile variable, the dynamic libraries can be used, but this requires the LEDA libraries to be available at run-time.

The makefile will create two programs, `doSim` and `checkParms`, when `make` is invoked with no parameters. `doSim` is the simulator itself and `checkParms` is a useful utility which will quickly check whether the parameter files are legal or not. The makefile can also create a utility called `createNet` which can be used to quickly generate a set of routes given a network topology. The user should be warned, however, that there is no guarantee that the routes will be provably dead-lock free. More details on the use of `createNet` is given in the `createNet.C` code file.

A.2 Introduction to Using the Simulator

The simplest way to use the simulator is to execute a single "run" by leaving the experiment parameter list empty. When the experiment parameter list contains no values, the simulator will run for the defined number of cycles using the values of the other parameters as set. A more complex use would be to set the simulator up to make several runs by entering one or more values into the experiment parameter list. As stated in A.3.2, an experiment parameter entry consists of the name of one of the other parameters followed by a range of values for that parameter. For each value in this range, the simulator executes a run with the defined number of total cycles. With two or more entries in the experiment parameter list, the simulator first executes a run for each value in the range specified in the last entry. Once these runs have been completed, the value of the second to last entry is advanced to the next value in its range, and again a run is executed for each value in the range specified in the last entry. This pattern continues until all combinations of values within all of the ranges have been used.

A.3 Configuring the Simulator

The simulator provides a powerful mechanism (particularly from the Programmer's point of view) for configuring a simulation. Three parameter files are used by the simulator: `network.dat`, `defSimParms.txt`, and `simParms.txt`. When the simulator runs through its initialization phase it first reads the file `network.dat`, which is used to specify the topology of the network including PE to PE message routes. The format of this file is discussed in Appendix A.3.1. The simulator then reads `defSimParms.txt`, and last it reads `simParms.txt`. These two files are used to state

the values of the parameters for a simulation. For historical reasons, the format of these two files is slightly different[2] and is discussed in Appendix A.3.2. The file `defSimParms.txt` specifies a default value for every parameter used by the simulator. The file `simParms.txt` can then be used to change the value of one or more parameters as needed (or none as the case may be). This system provides a good balance between flexibility and convenience as often a large number of parameters remain the same for many different topologies. Thus these parameters do not clutter up the individual `simParms.txt` files, which in turn reduces the number of typographic errors. An example of each of these files is included in the `sampleDataFiles` sub-directory in the release directory.

A.3.1 Format of `network.dat`[5]

This file specifies four types of information: the number of hosts (PEs) in the network, the number of switches in the network, the links among hosts and switches, and the routing table for PE to PE messages. The first line of the file is an integer specifying the number of hosts, and the second line is an integer specifying the number of switches. The links are then specified one link per line by indicating the two ports that the link connects. Port names for hosts and switches have the forms

Hx

Sy-p

respectively, where x is the number of the host, y is the number of the switch, and p is the number of the port on the switch. Hosts are numbered from zero to $n-1$ where n is the number of hosts in the network, switches are numbered from zero to $m-1$ where m is the number of switches in the network, and ports are numbered similarly based on the number of ports the switch is defined to have. Generally, hosts will be linked to switches, and switches can be linked to either hosts or switches. Two example link specifiers are:

```
H2 S4-3
S0-1 S2-6
```

This first specifier creates a link between Host 2 and Port 3 of Switch 4. The second creates a link between Port 1 of Switch 0 and Port 6 of Switch 2. As the links are bidirectional, it is not necessary (and is undesirable) to specify the same link twice by reversing the order of the endpoints.

[2]The format for the two data files should probably eventually be standardized, but it is not a particularly pressing issue.

IMPORTANT: Port 0 of each switch is reserved by the simulator for the Token Managers, so one should not use it to connect switches or hosts.

Finally, the `network.dat` file must contain a route specifier for every pair of hosts in the network, one route specifier per line. A route specifier has the form

Rs-d ℓ $p_1 p_2 \cdots p_\ell$

where s is the source host's number, and d is the destination host's number. ℓ is the length of the route measured in number of switches a message must pass through from the source host to the destination host. p_1 through p_ℓ are the port numbers through which the message must exit for each switch along its route. An example route specifier is:

```
R1-7 3 115
```

This example specifies that a message from host 1 to host 7 must pass through 3 switches. The first switch is exited through port 1, the second through port 1, and the third through port 5.

A complete file for specifying the four host, two switch network shown in Figure A.1 can be written as follows.

```
4
2
H0 S0-7
H1 S0-2
H2 S1-5
H3 S1-3
S0-4 S1-1
R0-0 1 7
R0-1 1 2
R0-2 2 45
R0-3 3 43
R1-0 1 7
R1-1 1 2
R1-2 2 45
R1-3 2 43
R2-0 2 17
R2-1 2 12
R2-2 1 5
R2-3 1 3
R3-0 2 17
```

```
R3-1 2 12
R3-2 1 5

R3-3 1 3
```

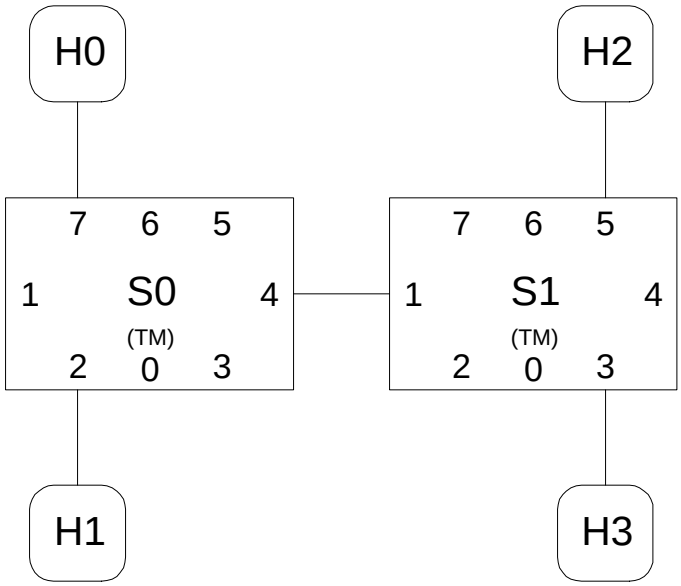


Figure A.1: **A sample network topology**

This is one possible configuration for a small network that has two switches and four hosts. For simplification, the TMs and SIUs are not shown, because the simulator sets up these connections automatically.

A.3.2 Format of `defSimParms.txt` and `simParms.txt`

A parameter specification in `simParms.txt` has the form

parameter_name: *parameter_entry*

The parameter names and their purposes are given in Appendix A.4 and the forms of a parameter entry are given below. This file may contain an arbitrary number of parameter specifications. The order in which the specifications appear does not matter, except for when there are multiple declarations for a given parameter name, in which case the last specification is used. A parameter specification in `defSimParms.txt` is similar, but the parameter name is left off. Thus a `defSimParms.txt` parameter specification has the form:

: *parameter_entry*

integer	→	$\{<digit>\}^+$ | `NOT_SET`
float	→	$\{<digit>\}^+$ $[.\{<digit>\}]$
string	→	$<alpha\text{-}numeric>$ $\{<alpha\text{-}numeric>$ | `-` | `_` | `.` $\}$
boolean	→	`true` | `t` | `false` | `f`
direction	→	`ascending` | `up` | `descending` | `down` | `random`
distribution	→	`uniform` | `neg_exp` | `exp` | `triangle`

Table A.1: **Formats of parameter types in EBNF notation**

This file contains one specification for every parameter in the simulator, and there is a required order. This order is the same order given in Appendix A.4. Because of the large number of parameters, it is recommended that the sample `defSimParms.txt` included in the code release be used as a "template".

A given parameter entry may consist of one or more values of one of the types listed in Table A.1. This table also describes the format for each value type. The simplest form for a parameter entry is a single value. For example:

```
traceFileName:  trace.txt
numSections:  32
```

Some parameters require an unordered list of values, which is written in the form

$[e_1, e_2, e_3, \ldots]$

Empty lists and single value lists are valid entries. Examples of this form are:

```
traceSections:  []
specialSections:  [0, 3, 27]
```

In these examples there are no trace sections and three special sections. The three special sections are section 0, section 3, and section 27.

The third form of a parameter entry is an indexed list of values. This form is used to assign a separate value to each index from 0 to n, where n is set by the given parameter. For example, the parameter that specifies the probability that an Isotach message to be sent is a READ, must be specified for each PE. Thus n for this parameter would be the number of PEs in the network. In general the format is one or more of

<indices> value

Because the number of indices can be quite high and often the value for many indices are the same, there are several ways of specifying the indices. The indices can be specified as a comma separated list of numbers, a range of numbers in the form x_0-x_1, or a combination thereof. Note that x_1 must be larger than x_0, and a value must be specified for every index from 0 to n. Some examples are:

```
peSendReadPct:  <0, 5> 0.1 <1-4> 0.2
siuCycles:  <0, 2-4, 7> 1 <1> 10 <6> 3
```

In the first example there are six PEs. There is a 10 percent chance that an Isotach message sent from PE 0 and PE 5 will be a READ, and there is a 20 percent chance for the other PEs (1 through 4). In the second example, there are eight SIUs, and the base cycles are set to 1 for SIU 0, SIUs 2 through 4, and SIU 7. SIU 1 has a base cycle value of 10, and SIU 6 has a value of 3.

The number of nodes may be different from one topology to the next, but it would be convenient to have a single set of defaults (e. g., in `defSimParms.txt`) used by all the topologies. For this reason, there are two special index entries: `<all>` and `<rest>`. The entry `<all>` stands for all possible index values 0 to n whatever n might be in the given simulation. If `<all>` is used, it must be the only index entry. The entry `<rest>` stands for all possible index values not already specified. If `<rest>` is used, then it must be the last entry. Some examples of the use of these two entries are:

```
peSendWritePct:  <all> 0.3
tmCycles:  <0, 2-4> 1 <rest> 3
```

In the first example, there is a 30 percent chance that an Isotach message will be a WRITE for all PEs, no matter how many PEs are in the given simulation. In the second example, the base cycles value of TM 0 and TMs 2 through 4 is set to 1, while the rest of the TMs, however many there may be, will have base cycles set to 3.

Finally, the parameter used for listing the experimental parameters (see Appendix A.2 has a special format that takes a list of other parameters. Currently, the only types of parameters allowed in this list are those that take on either numeric or boolean values. Each parameter in this list is formatted similarly to the regular parameters as described above: (*parameter_name*: *parameter_entry*) but there are a couple differences in the parameter entry. If the parameter normally

requires indexed entries, then its parameter entry must consist of one and only one index specification formatted normally. However, all possible index values do not need to be specified. All parameters must specify a start value. If the parameter takes numeric values, an end value and an interval must also be specified. If the parameter takes boolean values, only the start value is required as the end value is implicitly the start value negated. Thus the format of the experiment parameter can be written as follows, where bold indicates that that value may or may not be present depending upon the type of the parameter:

parameter_name: <***indices***> *start_value* ***end_value interval***

In the first of the following examples, five runs will be executed. In the first run, the task cycles for Token Handler 0 and Token Handlers 2 through 4 will be set at 10. On the second run, the task cycles for each Token Handler will be set at 20. In the second example, a total of 60 runs will be executed. As described in Appendix A.2, first the value of the `peCycles` parameter is modified. After these six runs have been executed, the `peCycles` value is reset to 500, `tokenHandlerCycles` is incremented to 6, and the `peCycles` values are modified again. Finally, after an execution for each of the `tokenHandlerCycles` values, `siuGeneratesToken` value "increments" to true, and the whole process is started again.

```
expParmList:  [tokenHandlerTaskCycles:  <0, 2-4> 10 50 10]
expParmList:  [siuGeneratesToken:  false,
               tokenHandlerTaskCycles:  <all> 1 21 5,
               peCycles:  <0, 2> 500 1000 100]
```

A.4 The Parameters

The following is a list of all the parameters that can be specified in the `simParms.txt` parameter file with an explanation of what each parameter is for. The order is presented in the required order for the `defSimParms.txt` parameter file. Some parameters are marked as <*disabled*>; although these must be set in `defSimParms.txt`, they do not have any function in the current simulator. Most of the <*disabled*> parameters are simply placeholders for possible future enhancements.

Identification

`simulationName` (string) — identifies a particular simulation to help distinguish it from other simulations.

`appendSimName` (boolean) — true means the `simulationName` value should be appended to the following filenames automatically.

File Names of Output

`statsFileNames` (string unordered list) — the names of the files in which the result statistics are printed. Most of the coded experiments only use one file, but it can be useful to separate some the printed statistics.

`traceFileName` (string) — the name of the file in which the output from the trace sections (if any) is printed. This file will not be created if there are no trace sections. See `printNetSnapshot`, `printSiuSnapshot`, and `traceSections`.

`portsFileName` (string) — the name of the file in which the memory addresses for each port in the network is printed. This file is for debugging purposes, and will not be created if `printPorts` is false. See `printPorts`.

`statusFileName` (string) — the name of the file in which the status of a running simulation is printed. This file will not be created if `printStatus` is false. See `printStatus` and `statusInterval`.

`specialFileName` (string) — the name of the file in which a secondary set of statistics may be printed. This file will not be created if there are no special sections. See `printPulseOverTime`, `printEpochOverTime`, `printIsBarrierOverTime`, and `specialSections`.

Output

`parmsToPrint` (string unordered list) — the parameters whose current values (along with the parameter names) should be printed in the stats file. These values are printed preceding the statistics for each run.

`printNetSnapshot` (boolean) — true means the current state of the Myrinet level network should be printed in the trace file. This data will only be printed for the sections specified as trace sections (see `traceSections`). The network snapshot gives a port by port rundown for each message in the network, including placement of flits within links and number of flits in buffers.

`netSnapshotInterval` (integer) — the interval of cycles (within a trace section) for which a network snapshot is printed. (i. e., a value of 1 would produce a snapshot every cycle).

`printSiuSnapshot` (boolean) — true means the current state of the SIU (Isotach level) should be printed in the trace file. This data will only be printed for the sections specified as trace sections (see]CodetraceSections). The SIU snapshot gives a machine by machine rundown of the SIU, including which messages are being forwarded to/from which machine.

`printQueueInfo` (boolean) — <*disabled*>

`printPorts` (boolean) — true means the memory addresses for all the ports in the network should be printed in the ports file. This parameter is intended for debugging purposes.

`printStatus` (boolean) — true means an indicator for how far the simulation has progressed should be printed in the status file.

`statusInterval` (integer) — the number of sections that should pass before a status indicator is printed in the status file (if `printStatus` is true).

Special Stats Printing

`printPulseOverTime` (boolean) — true means the actual pulse length (token turn-around time) should be printed for every measured pulse length during a special section.

`printEpochOverTime` (boolean) – true means the actual epoch length (barrier completion time) should be printed for every measured epoch length during a special section.

`printIsBarrierOverTime` (boolean) — true means that a chronological list of whether a token was a barrier or not should be printed for every token received during a special section.

Base Message Lengths

`ShRefLength` (integer) — the length in flits of a shared reference (i. e., an Isotach message) not counting routing flits.

`nonIsoLength` (integer) — the length in flits of an non-Istoach message not counting routing flits.

`tokenLength` (integer) — the length in flits of a token not counting routing flits.

`isochronMarkerLength` (integer) — the length in flits of an isochron marker not counting routing flits.

`eopMarkerLength` (integer) — the length in flits of an EOP marker not counting routing flits.

`signalMarkerLength` (integer) — the length in flits of a signal marker not counting routing flits.

`barrierMarkerLength` (integer) — the length in flits of a barrier marker not counting routing flits.

Section Information

`numSections` (integer) — the total number of sections in a run.

`discardFirstSection` (boolean) — true means the data from the first section should not be included in the statistics computations.

`traceSections` (integer unordered list) — the sections that are trace sections. Data from the trace sections are not included in the statistics computations. Sections are numbered starting at 0.

`specialSections` (integer unordered list) — the sections that are special sections. Data from the special sections are not included in the regular statistics computations, but may be included in the secondary statistics computations. See `printPulseOverTime`, `printEpochOverTime`, and `printIsBarrierOverTime`. The secondary statistics tend to be very expensive in terms of execution length and amount of output.

`discardSectionLength` (integer) — how many cycles long is the first section of the simulation if `discardFirstSection` is true.

`regularSectionLength` (integer) — how many cycles long are the regular sections (from which the statistics are computed).

`traceSectionLength` (integer) — how many cycles long are the trace sections. Note that if the first section is a trace section and `discardFirstSection` is true, `discardSectionLength` takes precedence.

`specialSectionLength` (integer) — how many cycles long are the special sections. Note that if a special section is also a trace section or discard section, `traceSectionLength` and `discardSectionLength` take precedence.

Statistical Distributions

`dustFactorThreshold` (float between 0 and 1) — the threshold that determines the range of values that cause the dust factor to be -1, 0, or 1. The dust factor is computed from a random value with a triangle distribution between 0 and 1. If the random value is less than `dustFactorThreshold`, then the dust factor is -1. If the random value is more than $1 -$ `dustFactorThreshold`, then the dust factor is 1. Otherwise, the dust factor is 0.

Token and Barrier Configurations

`siuGeneratesToken` (boolean) — true means that the SIU will generate an initial token causing two tokens to be in the TM→SIU→TM loop.

`isBarrierIsotach` (boolean) — true means that the Isotach barrier algorithm is enabled. False means that the non-Isotach simple centralized barrier algorithm is enabled.

Order of Elements in Simulation Execution

`peDoCycleOrder` (direction) — the order of the PEs for which the PE actions are executed for each cycle in the simulation. The PEs are ordered by node number. Random means that the order changes each cycle.

`siuDoCycleOrder` (direction) — the order of the SIUs for which the SIU actions are executed for each cycle in the simulation. The SIUs are ordered by node number. Random means that the order changes each cycle.

`tmDoCycleOrder` — the order of the TMs for which the TM actions are executed for each cycle in the simulation. The TMs are orered by node number. Random means that the order changes each cycle.

`tmSendTokenOrder` — the order of the destinations for each token wave from a TM. Ascending means that the SIUs (low node number to high node number) are sent tokens before the neighboring TMs (also low node number to high node number). Descending means that the TMs (high node number to low node number) are sent to before the SIUs (high node number to low node number). Random means that the order changes for each token wave.

Network Delays

`nearPropDelay` (integer) — the length in flits of the near links.

`farPropDelay` (integer) — the length in flits of the far links.

`xbarPropDelay` (integer) — the length in flits of the switch crossbar.

`nearSignalGen` (integer) — the length in cycles that it takes a near buffer (downstream buffer of a near link) to generate the STOP/GO signal to be sent upstream. This value is not to be confused with the propagation delay of the near link.

`farSignalGen` (integer) — the length in cycles that it takes a far buffer to generate the STOP/GO signal to be sent upstream. This value is not to be confused with the propagation delay of the far link.

`xbarSignalGen` (integer) — <*disabled*>

`nearBufKgVal` (integer) — the low watermark for near buffers. A GO signal is generated and sent if the port is currently STOPped and the number of flits drops below this value. See the Myrinet Specification for more information about the kg value.

`farBufKgVal` (integer) — the low watermark for far buffers.

`xbarBufKgVal` (integer) — <*disabled*>

`nearBufHVal` (integer) — the number of hysteresis flits for the near buffers. The high watermark is the kg value plus the h value. A STOP signal is generated and sent if the the port is

currently GOing and the number of flits reaches the high watermark value. See the Myrinet Specification for more information about the h value.

`farBufHVal` (integer) — the number of hysteresis flits for the far buffers.

`xbarBufHVal` (integer) — <*disabled*>

`nearBufCapacity` (integer) — <*disabled*>

`farBufCapacity` (integer) — <*disabled*>

`xbarBufCapacity` (integer) — the maxmimum number of flits a crossbar buffer (a switch's internal buffer downstream of the crossbar) can hold. When the number of flits reaches this value, the flits start to back up in the crossbar.

Node Base Cycles

`netCycles` (integer) — the interval of simulation cycles that correspond to a single Myrinet network action. (i. e., a value of one means that message flits are advanced every cycle, and a value of two means that message flits are advanced every other cycle.)

`peCycles` (integer indexed list) — the interval of simulation cycles that correspond to a single PE action. The indices range from 0 to the number of PEs − 1 (i. e., number of hosts − 1).

`siuCycles` (integer indexed list) — the interval of simulation cycles that correspond to a single SIU action. The indices range from 0 to the number of SIUs − 1 (i. e., number of hosts − 1).

`tmCycles` (integer indexed list) — the interval of simulation cycles that correspond to a single TM action. The indices range from 0 to the number of TMs − 1 (i. e., number of switches − 1).

Token Manager

`tmPulseDelay` (integer indexed list) — the number of cycles a TM waits between receiving the last token from a neighbor and starting to send the next token wave. The indices range from 0 to the number of TMs − 1 (i. e., number of switches − 1).

SIU Machine Task Cycles

`fromHostFilterTaskCycles` (integer indexed list) — the number of task cycles the From Host Filter takes to complete a single task. This value is measured in units of SIU base cycles (i. e., 1 task cycle = `siuCycles`). The indices range from 0 to the number of SIUs − 1 (i. e., number of hosts − 1).

`measurerTaskCycles` (integer indexed list) — the number of task cycles the Measurer takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`senderTaskCycles` (integer indexed list) — the number of task cycles the Sender takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`toNetMergerTaskCycles` (integer indexed list) — the number of task cycles the To Net Merger takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`fromNetFilterTaskCycles` (integer indexed list) — the number of task cycles the From Net Filter takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`localStreamMergerTaskCycles` (integer indexed list) — the number of task cycles the Local Stream Merger takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`clonerTaskCycles` (integer indexed list) — the number of task cycles the Cloner takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`tokenHandlerTaskCycles` (integer indexed list) — the number of task cycles the Token Handler takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`extractorTaskCycles` (integer indexed list) — the number of task cycles the Extractor takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`sorterTaskCycles` (integer indexed list) — the number of task cycles the Sorter takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`eopBuilderTaskCycles` (integer indexed list) — the number of task cycles the EOP Builder takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`isoMergerTaskCycles` (integer indexed list) — the number of task cycles the Iso Merger takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

`toHostMergerTaskCycles` (integer indexed list) — the number of task cycles the To Host Merger takes to complete a single task. The indices range from 0 to the number of SIUs − 1.

SIU Machine Alternate Task Cycles

`fromHostFilterAltTaskCycles` (integer indexed list) — the number of task cycles the From Host Filter takes to complete a single task when its alternate task time is in effect. See `fromHostFilterAltTaskPct`. The indices range from 0 to the number of SIUs − 1 (i. e., number of hosts − 1).

`measurerAltTaskCycles` (integer indexed list) — the number of task cycles the Measurer takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`senderAltTaskCycles` (integer indexed list) — the number of task cycles the Sender takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`toNetMergerAltTaskCycles` (integer indexed list) — the number of task cycles the To Net Merger takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`fromNetFilterAltTaskCycles` (integer indexed list) — the number of task cycles the From Net Filter takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`localStreamMergerAltTaskCycles` (integer indexed list) — the number of task cycles the Local Stream Merger takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`clonerAltTaskCycles` (integer indexed list) — the number of task cycles the Cloner takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`tokenHandlerAltTaskCycles` (integer indexed list) — the number of task cycles the Token Handler takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`extractorAltTaskCycles` (integer indexed list) — the number of task cycles the Extractor takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`sorterAltTaskCycles` (integer indexed list) — the number of task cycles the Sorter takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`eopBuilderAltTaskCycles` (integer indexed list) — the number of task cycles the EOP Builder takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`isoMergerAltTaskCycles` (integer indexed list) — the number of task cycles the Iso Merger takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

`toHostMergerAltTaskCycles` (integer indexed list) — the number of task cycles the To Host Merger takes to complete a single task when its alternate task time is in effect. The indices range from 0 to the number of SIUs − 1.

Likelihood of SIU Machine Alternate Task Cycles

`fromHostFilterAltTaskPct` (float indexed list) — the probability that each task's completion time in the From Host Filter will be set to `fromHostFilterAltTaskCycles` instead of `fromHostFilterTaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`measurerAltTaskPct` (float indexed list) — the probability that each task's completion time in the Measurer will be set to `measurerAltTaskCycles` instead of `measurerTaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`senderAltTaskPct` (float indexed list) — the probability that each task's completion time in the Sender will be set to `senderAltTaskCycles` instead of `senderTaskCycles`. The indices range

from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`toNetMergerAltTaskPct` (float indexed list) — the probability that each task's completion time in the To Net Merger will be set to `toNetMergerAltTaskCycles` instead of `toNetMerger-TaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`fromNetFilterAltTaskPct` (float indexed list) — the probability that each task's completion time in the From Net Filter will be set to `fromNetFilterAltTaskCycles` instead of `fromNet-FilterTaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`localStreamMergerAltTaskPct` (float indexed list) — the probability that each task's completion time in the Local Stream Merger will be set to `localStreamMergerAltTaskCycles` instead of `localStreamMergerTaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`clonerAltTaskPct` (float indexed list) — the probability that each task's completion time in the Cloner will be set to `clonerAltTaskCycles` instead of `clonerTaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`tokenHandlerAltTaskPct` (float indexed list) — the probability that each task's completion time in the Token Handler will be set to `tokenHandlerAltTaskCycles` instead of `tokenHandler-TaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`extractorAltTaskPct` (float indexed list) — the probability that each task's completion time in the Extractor will be set to `extractorAltTaskCycles` instead of `extractorTaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`sorterAltTaskPct` (float indexed list) — the probability that each task's completion time in the Sorter will be set to `sorterAltTaskCycles` instead of `sorterTaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`eopBuilderAltTaskPct` (float indexed list) — the probability that each task's completion time in the EOP Builder will be set to is `eopBuilderAltTaskCycles` instead of `eopBuilderTask-`

`Cycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`isoMergerAltTaskPct` (float indexed list) — the probability that each task's completion time in the Iso Merger will be set to `isoMergerAltTaskCycles` instead of `isoMergerTaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

`toHostMergerAltTaskPct` (float indexed list) — the probability that each task's completion time in the To Host Merger will be set to `toHostMergerAltTaskCycles` instead of `toHostMerger-TaskCycles`. The indices range from 0 to the number of SIUs − 1. Each value must be between 0 and 1.

Buffer Sizes

`peReceiveBufSize` (integer indexed list) — <*disabled*>

`toHostFilterBufSize` (integer indexed list) — <*disabled*>

`toNetFilterBufSize` (integer indexed list) — <*disabled*>

`hostFilterToMeasurerBufSize` (integer indexed list) — <*disabled*>

`measurerToSenderIsochronBufSize` (integer indexed list) — <*disabled*>

`measurerToSenderMarkerBufSize` (integer indexed list) — <*disabled*>

`senderToNetMergerBufSize` (integer indexed list) — <*disabled*>

`senderToTokenHandlerBufSize` (integer indexed list) — <*disabled*>

`netFilterToTokenHandlerBufSize` (integer indexed list) — <*disabled*>

`tokenHandlerToSenderBufSize` (integer indexed list) — <*disabled*>

`tokenHandlerToHostMergerBufSize` (integer indexed list) — <*disabled*>

PE Message Sending

`peSendNonIsoPct` (float indexed list) — the probability that a message to be sent is non-Isotach. The indices range from 0 to the number of PEs − 1. Each value must be between 0 and 1. `peSendNonIsoPct` + `peSendIsochronPct` should be $\not> 1$.

`peSendIsochronPct` (float indexed list) — the probability that a message to be sent is the start of an Isochron. The indices range from 0 to the number of PEs − 1. Each value must be between 0 and 1. `peSendNonIsoPct` + `peSendIsochronPct` should be $\not> 1$.

`peSendReadPct` (indexed float list) — the probability that a message to be sent, if part of an Isochron, will be a READ. The indices range from 0 to the number of PEs − 1. Each value must be between 0 and 1. `peSendReadPct` + `peSendWritePct` should be $\not> 1$.

`peSendWritePct` (float indexed list) — the probability that a message to be sent, if part of an Isochron, will be a WRITE. The indices range from 0 to the number of PEs − 1. Each value must be between 0 and 1. `peSendReadPct` + `peSendWritePct` should be $\not> 1$.

`peIsochronDistr` (distribution indexed list) — the distribution used to determine the number of messages that will be in an Isochron. The indices range from 0 to the number of PEs − 1.

`peMinShRefs` (integer indexed list) — the minimum number of messages that will be in an Isochron. The indices range from 0 to the number of PEs − 1.

`peMaxShRefs` (integer indexed list) — the maximum number of messages that will be in an Isochron. The indices range from 0 to the number of PEs − 1.

`peMeanShRefs` (integer indexed list) — the mean number of messages that will be in an Isochron. The indices range from 0 to the number of PEs − 1.

Other PE parameters

`nonIsoBarrierCentralPe` (integer) — the node number of the PE that will act as the central PE for the non-Isotach barrier algorithm if `isBarrierIsotach` is false.

`peDestOrder` (direction) — the order in which a PE chooses its destination when it sends a message. Ascending means a PE will always send to the PE with the next node number greater than its own (the PE with the highest node number sends to PE 0 of course). Descending means a PE will always send to the PE with the next node number lower than its own (PE 0 will always send to the PE with the highest node number). Random means that the destination is picked randomly for each send.

`peSendDelta` (integer indexed list) — <*disabled*>

`verifyBarrierCompletion` (boolean) — true means that the internal checking for the barrier algorithms is turned on.

Experiments

`expNum` (integer) — this value determines which set of statistical output will be generated.

`expParmList` (special list) — For the format and use of `expParmList`, see Appendices A.2 and A.3.2.

A.5 Experiments and Output

Throughout the course of each simulation, statistical data is generated and output to the file(s) specified in the `statsFileNames` parameter. There are a very large number of different statistics that can be generated, so the `expNum` parameter is used to determine which set of statistics should be printed. If `expNum` is set to 0, a default set will be used. Unfortunately, precisely because of the large number of possible statistics, any pre-defined format of output is likely to be of limited value to the user's actual needs. This means that some programming will probably have to be done by the user. See Section 4.2 for a discussion of possible enhancements to this area of the simulator.

The current implementation has hooks for the default output set and four experiments. As currently written, the default selection prints end of run statistics for pulse lengths (token turn-around times) and epoch lengths (barrier completion times). The first experiment prints statistics used to compare link latencies and blocked times with pulse and epoch lengths. In particular, this experiment is useful in comparing one token in the TM→SIU→TM() loop (SIUs do not generate an initial token) with two tokens in the TM→SIU→TM loop (SIUs generate an initial token). This experiment prints a more complete set of statistics than the default. Experiment two is used to examine the effect of increasing the SIU token turn-around delay on pulse lengths for both one token and two tokens in the TM→SIU→TM loop. This experiment is intended to have two experiment parameters: first `siuGeneratesToken` and second `tokenHandlerTaskCycles`. Two data files are generated, one for measurements taken at the TMs and the other for measurements taken at the SIUs. At the end of each run, only the average pulse length is printed, each value separated by a tab. A new line is started when the first parameter is changed. Experiment three is unused, and experiment four is used for examining the actual epoch lengths over a period of time.

The default output can be edited in the function `CObserver::PrintDefault()`, and the experiment output can be edited in `CObserver::PrintExp`x`()` where x is the experiment number. Additional experiments can be easily added by creating another entry in the case statement in `CObserver::Print()` and writing an appropriate `CObserver::PrintExp`x`()` function. These printing functions are called at the end of every section simulated, so intermediate data can be printed. The end of run can be detected by comparing the current section to the total number of (statistics gathering) sections. This is demonstrated in the code. For more information about specific objects and functions used in computing and printing the statistics, see Appendix B.5.

A.6 Statistical Accuracy

To get a statistically stable population, one needs at least 30 separate data points. This is achieved by making a run consist of at least 30 sections, since the simulator gathers measurements (such as averages and standard deviations for pulse lengths) separately for each section. In addition to these data sections, a discard section should also be included at the start of a run. The discard section is used to skip spurious data that may occur before the simulator acquires a stable state. Given measurements that are supposed to have a stable value across a large period of time, to determine what the length of the sections should be (all data sections should have the same length), one should perform some sensitivity studies. Choose a fixed number of cycles for each section, execute a complete run, and analyze the standard deviation of the measurements across runs. In this way, one can compare the similarity of results between sections. One can set a threshold of similarity (for example 0.01) for which increasing section lengths can be tried, until the cross-section standard deviation is less than or equal to the threshold. In this way, one can comfortably state that each section is long enough to exhibit stable behavior. One practical tip for this process is to try increasing only the discard section length first as sometimes the standard deviation can be thrown off by one (in this case spurious) value.

Appendix B

Programmer's Guide

B.1 Structural Overview

The simulator can conceptually be broken into four major parts: the parameter mechanism, the Myrinet level of the actual simulator, the Isotach level of the simulator, and the data gathering and statistics mechanism. Each of these parts is discussed in more detail below. There are three file pairs (header file and source code file) that contain globally scoped variables, constants, functions, and type definitions. The fundamental global files, `globals.H` and `globals.C`, contain the globals that are basic to every other data structure in the program. For example, most of the type definitions and constants, as well as the random number generator and distribution functions are contained within these files. The files `parmGlobals.H` and `parmGlobals.C` contain globals that are only pertinent to the parameter mechanism. These files are discussed in more detail in Appendix B.2. Finally, the files `simGlobals.H` and `simGlobals.C` contain globals that are only pertinent to certain parts of the simulator, including the global parameter handler, the global observer, and the global clock. Additionally, these two files set up a rudimentary form of exception control — namely catch any fatal signals, and "crash" the program with a marginally useful error message and the clock cycle in which the error occurred.

The global parameter handler and global observer (along with a number of other objects in the program) follow the Open/Close model. Although the objects are created statically, there is no way to guarantee the order the objects are created, so initialization (and cleaning up) is controlled through the use of the `Open()` (and `Close()`) functions that these objects provide. More details on each of these objects are given below in B.2 and B.3 respectively.

The `main()` function is located in `doSim.C`. After initialization the main loop of the program is executed in the function `doSim`. The outermost loop controls the number of runs that are executed by calling `CParmHandler::GoToNextExp()` which advances the experiment parameters (if any) and returns whether or not all experiments have been executed. For each run, a `CMyrinetNetworkArbt` object is created. It is this class (actually its base class `CMyrinetNetwork`) that consists of or maintains all the other simulator objects including nodes, ports and messages. The inner loops of `doSim` loop through sections and cycles. The simulation at each cycle is implemented through a two-phase system. First, the `doCycle` function of the network (and respectively all its components) is called. It is through this function that the bulk of the processing for a given cycle occurs. However, due to the timestepped nature of the simulator, it is important that changes to one component at timestep n do not affect other components until timestep $n + 1$. Thus, a second function `CommitChanges` is called to activate any changes made by one component that directly affect another component.

B.2 Parameters

The simulator interacts with the parameter mechanism through one object, the global parameter handler of type `CParmHandler`. From the simulator's point of view, all that needs to be done is use one of the many access functions defined for `CParmHandler` to get the value or values of a names parameter. The complicated part actually occurs during initialization, when all the parameters are read in from the parameter files. `CParmHandler` stores the parameter data in several different ways. First, certain important parameters such the number of hosts and number of switches are stored directly. Included in this set of parameters are `traceSections` and `specialSections` since these are frequently accessed and require some special handling (they are sets). The second type of storage is for the `expParmList`. This is a special (and somewhat complicated) parameter, so the handling of it is relegated to another class which is discussed below: `CExpParmHandler`.

Finally, the rest of the parameters are stored in a hashed array. This is actually acomplished using two h_arrays (LEDA hashed array data type), both keyed on the parameter name in the form of a string. The first h_array associates a parameter name with a *type*, which is encapsulated in the `CParmType` class. The valid types are defined in the `VarTypeT` enumerated type (in `parmGlobals.H`). A given `CParmType` also has an associated `ListTypeT` (enumerated in `parmGlobals.H`) which indicates whether the parameter should consist of a single value or should consist of a list, the size

of which is either the number of hosts or the number of switches within the network. The second h_array associates a parameter name with the parameter's value(s) via a pointer to `CParm` object. `CParm` is a base class with subclasses consisting of instantiations of `TTypedParm`, a template that encapsulates the functionality of reading and storing a parameter value or values of "arbitrary" type. Access to these values is through polymorphic function calls. The `CParmType` map is used to determine which instantiation is used when reading values from the parameter files, but the object's pointer is downcast to a `CParm` pointer so all the parameters can be stored in one h_array and accessed identically. Unfortunately, there are occasions when the type of the parameter is needed in some `TTypedParm` methods, which breaks down the polymorphic model somewhat. To get around these problems, a class called `CVarBox` is used. This class can store a value of any type defined for `CParmType` and additionally can return what its current type is. This class should be exercised with caution however, as it does bypass the normal strong C++ compile-time type-checking (in favor of slower and weaker run-time checking).

The `parmGlobals.H` and `parmGlobals.C` files also contain functions that are used to facilitate reading parameters from parameter files as well as convert strings to the types defined in `VarTypeT`.

`CExpParmHandler` is used to handle the more complicated `expParmList` parameter. Since `expParmList` can consist of several entries, each of which is a parameter with associated values, `CExpParmHandler` uses a separate class, `CExpParm`, to encapsulate each entry. Each `CExpParm` has the parameter name and type of the corresponding parameter in `CParmHandler` that should be modified. Each `CExpParm` is also capable of storing a list of targets corresponding a list of nodes (if defined for the given parameter), a start value, and end value, and an interval value. These last three are stored as `CParm` pointers to facilitate comparisons with the current values of the corresponding parameter (in `CParmHandler`). `CExpParmHandler` provides functions that are used to advance from one experiment to the next (it contains a pointer to `CParmHandler` and can modify the current value of a parameter through this pointer) which return a boolean value indicating whether the advance was successful (i. e. the last experiment was performed).

B.3 Myrinet Level

The Myrinet level, which consists of the modeling of messages moving within the network (including switches), is perhaps the most confusing part of the entire program as it is the part that contains

all of the legacy code. The `CMyrinetNetwork` class, which is inherited from earlier simulations originally consisted of only certain types of nodes in the network, but was modified so that all nodes appear as arrays within the class, including those nodes that are in large part treated as part of the Isotach level. Unfortunately, nothing is ever so clearly defined — there is a certain amount of crossover between the levels (e. g. the `CTokenManager` is an Isotach level object, but does some operations at the Myrinet level). However, in general the Myrinet level classes are all those named `CMyrinet...`, as well as `CRoute` and `TPortQueueElem`.

Due to previous simulator design, the Myrinet level part of the simulation is "message-centric" in that each cycle, it updates each message as opposed to updating the ports through which those messages flow. Note that the messages referred to in the Myrinet level correspond to the `CMyrinetMessage` class and should not be confused with the Isotach level message `CIsoMessage` (in fact a `CMyrinetMessage` contains a `CIsoMessage` as its payload). Thus, the `CMyrinetNetwork` object, which contains a list of all messages within the network, calls the `Adjust()` function for each message in the list, which tells the message to move itself through the network. This adjustment is accomplished by advancing the message through a series of states based on the location of the head of the message as detailed in B.3.2. As stated in Section 3.3, from the message's point of view, the network consists of a series of alternating buffers and links across which the flits of a message are strung. Due to the history of the simulator these buffers and links are actually referred to in the code as input and output ports (`CMyrinetInputPort` and `CMyrinetOutputPort` objects) respectively. The input ports roughly correspond to buffers, because they can store some number of flits. The output ports roughly correspond to links, because they contain the bitstream[1] that represents the latency from one input port to the next. Depending upon the message state, `Adjust()` may call `ForwardOne()` which moves each flit in the message forward (if possible). The general mechanism for moving flits is described in Section 3.2, and the actual functions and objects involved are detailed in the following section (B.3.1).

[1] A 1 indicates that a flit from this message occupies the link at that point, and a 0 means that this message does not occupy the link at that point. Note that this does not necessarily indicate that the link is empty where there is a 0, because some other message may also occupy the same link (though necesarily in different bit positions).

B.3.1 Moving a Message's Flits[2]

Certain peculiarities arise due to the "message-centric" nature of this part of the simulation. Most notably, the perspective of where flits reside within the network is from a message's point of view. In other words, each buffer storage and link bitstream is stored on a per-message basis. This is accomplished through a class called `CMsgSpecPort` (for message specific port). A message contains an ordered list of `CMsgSpecPort` objects, each of which represents (i. e. has a pointer to) a particular `CMyrinetInputPort` or `CMyrinetOutputPort`. If the `CMsgSpecPort` represents a `CMyrinetOutputPort`, then it also contains a `CMyrinetLink` which represents its message's occupancy of that output port's link.

The function `CMyrinetMessage::ForwardOne()` traverses the message's list of `CMsgSpecPort` objects, starting with the entry at the head. The `MoveFlit()` member function of each entry is called to perform the flit changes that occur in that port (as well as it's upstream port) as described in Sections 3.2 and 3.3. The `MoveFlit()` function maintains flow control by calling the `CheckWatermark()` member function of a `CMyrinetInputPort` each time that port's number of flits is changed. The functions used to move flits within a link are `Proceed()` and `Pump` member functions of the `CMyrinetLink` object. `Proceed()` right-shifts a zero into the link bitstream, and `Pump()` right-shifts a one. Two special versions of these functions were created in the switch enhancement that allows for flits to be blocked within a link by taking a parameter indicating the blocked position (if any).

B.3.2 Message State Transitions

As a message traverses the network, it changes its state based on the location of the head flit. We refer to these changes as transitions. Figure B.1 through Figure B.5 show all the states and all the possible state transitions within the system. Each figure is a diagram that represents a different section of the Myrinet network through which a message may pass. The `ENTER_LINK` state, which indicates that the head of a message is in one of the inter-node links, is the common state in all the diagrams, so one may easily consider all the figures together as one large finite state system. There are three basic types of transitions, represented by the arrows in the diagrams, that can occur. First, the head of the message can move freely from one port to the next, usually indicating a change from one state to another. Second, the head of the message may be unable to move

[2]It is highly recommended that Section 3.2 be read prior to Appendix B.3.1.

forward because it is STOPped. When this occurs, the head of the message is in an input port that is not allowed to send any more flits downstream. This transition usually indicates changing to or remaining in one of the `PEND...LINK` states. Third, the head of the message may be unable to move forward because it is blocked by another message (i. e. contention). This can occur for a couple of reasons (either blocked from leaving an input port or blocked from entering an output port) and indicates changing to or remaining in one of the other `PEND...` states.

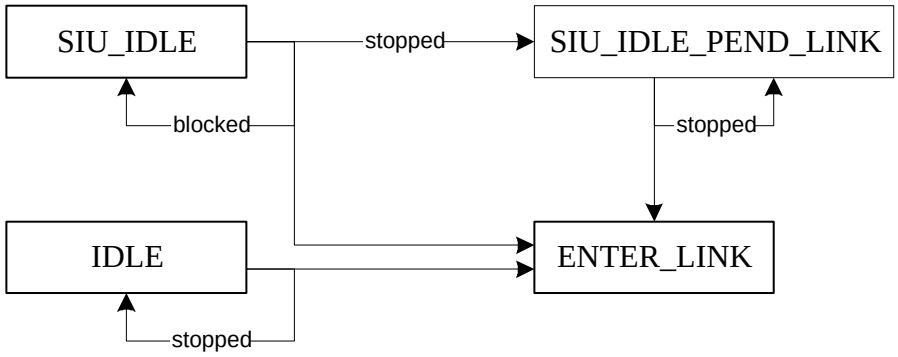


Figure B.1: **States of entry into the system**

Figure B.1 shows the possible entry points into the system. If a message is sent from either a `CProcElem` or a `CTokenManager`, the entire message is placed in that node's `outgobuffer` and starts in the `IDLE` state. If the head flit is free to move to the node's output port (i. e. the exiting inter-node link), then the message changes its state to `ENTER_LINK`. However, if the port has been STOPped, then the message remains in its `IDLE` state. If a message is sent from an SIU (either from the `CToHostMerger` or the `CToNetMerger`), the entire message is placed in that node's `outgobuffer` and starts in the `SIU_IDLE` state. Here, a message can be free to enter the link or either STOPped or blocked from entering the link by another message that is traversing the SIU bypass and entered the link first. If the new message is free to enter the link, then it changes its state to `ENTER_LINK`. If the new message is STOPped it changes its state to `SIU_IDLE_PEND_LINK`, but if it is blocked it remains in the `SIU_IDLE` state. Finally, a message in the `SIU_IDLE_PEND_LINK` state will remain in that state as long as it is STOPped. As soon as it can go, the head flit enters the link and the message changes to the `ENTER_LINK` state.

Figure B.2 shows the possible exit points from the system. When the head flit of a message arrives at the input port of a `CProcElem`, the message changes its state to `ARRIVE_PE`. The message remains in this state until all the flits have entered the input port, at which point the message is

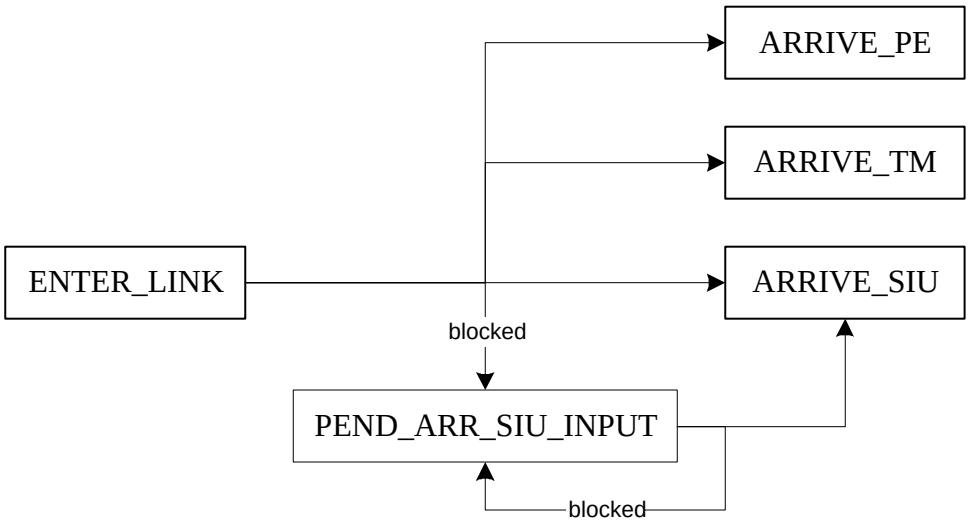


Figure B.2: **States of exit from the system**

removed from the Myrinet level. A message that arrives at a `CTokenManager` behaves the same way, although in this case the state is `ARRIVE_TM`. Similarly, a message that is to be received by an SIU enters the `ARRIVE_SIU` state and remains there until all the flits have arrived. However, at the SIU it is possible for a message to be blocked by a previous message traversing the SIU bypass. In this case, the arriving message must enter the `PEND_ARR_SIU_INPUT` state, where it remains until unblocked. Furthermore, some messages arriving at an SIU are not received by the SIU and instead traverse the SIU bypass. This is shown in Figures B.3 and B.4.

Due to the large number of states involved in the SIU bypass, this section has been split into two diagrams. Figure B.3 shows the process of entering the SIU bypass and ends in the `ENTER_SIU_LINK` state, which indicates that the head flit is in the link connecting the two sides of the SIU. Figure B.4 starts at the `ENTER_SIU_LINK` state and shows the process of exiting the SIU bypass. Starting with the `ENTER_LINK` state in Figure B.3, the head flit of the message enters the SIU's input port (either the host or the network side depending upon the direction of travel). As described before, if another message is already resident in this port, the arriving message changes its state to `PEND_ARR_SIU_INPUT`, where it remains until it is free to go. If the message is the first or only resident in the input port, its state changes to `ARRIVE_SIU`. However, unlike above, the message then tries to enter the SIU bypass link. If successful, it state changes to `ENTER_SIU_LINK`. However, if it is then blocked by another message traversing the link (which in practice happens infrequently), it changes to the `PEND_ARR_SIU_OUTPUT` state. If STOPped (either immediately or

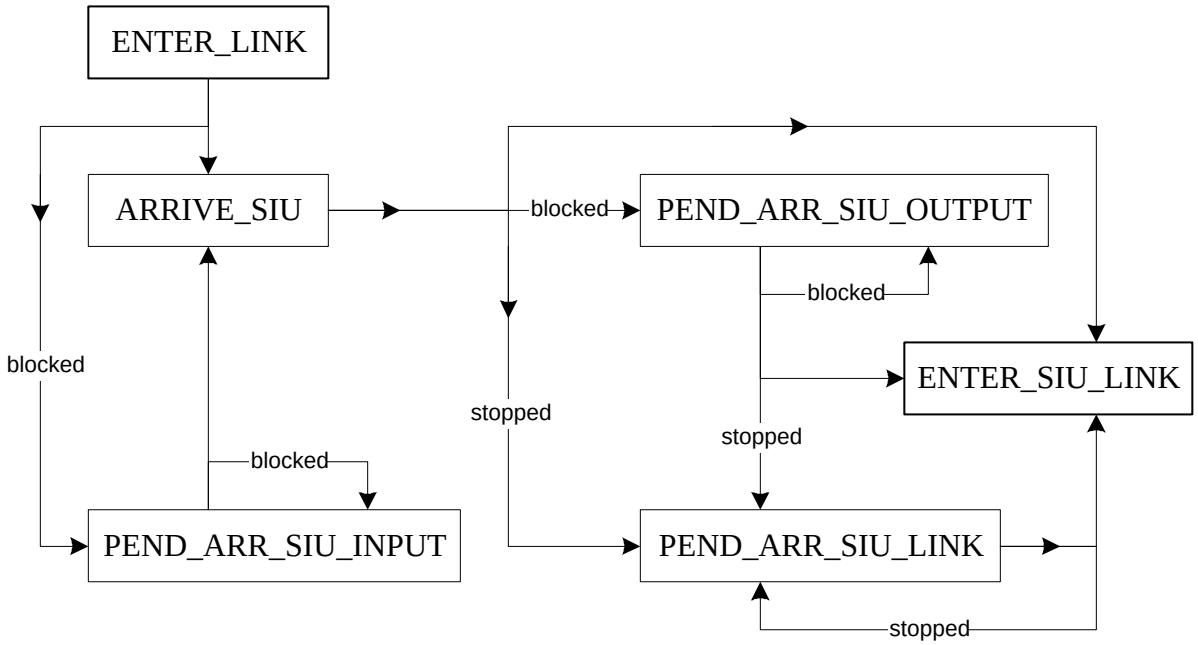


Figure B.3: **States involved in entering the SIU bypass**

after having been blocked) it changes to the `PEND_ARR_SIU_LINK` state, where it remains until it is free to enter the link.

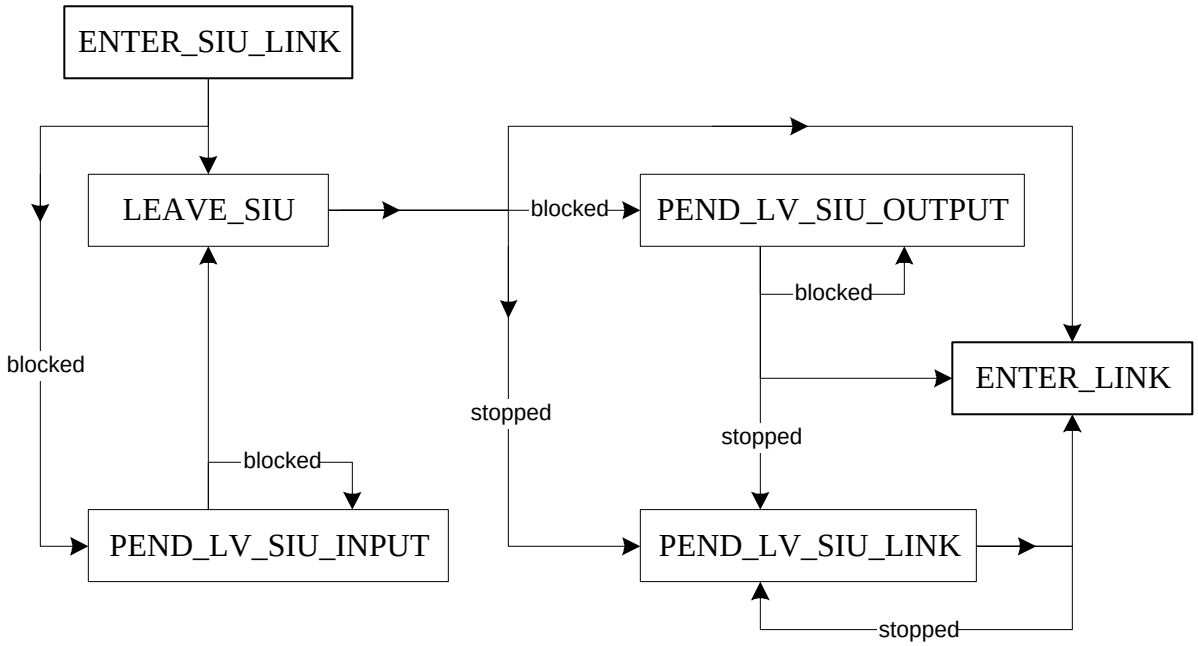


Figure B.4: **States involved in exiting the SIU bypass**

In Figure B.4 we start with the head flit exiting from the SIU bypass. The head flit enters the buffer, and if it is free to go, then the message changes to the `LEAVE_SIU` state. However, if a previous message (from the bypass) is still exiting (i. e. it was STOPped earlier and the flits backed up), the arriving message must block, and its state changes to `PEND_LV_SIU_INPUT`. Once the message is the first or only message from the bypass, it may have to contend with messages being sent from the SIU, in which case its state changes to `PEND_LV_SIU_OUTPUT`. If the message is STOPped (again either immediately or after having been blocked) its state changes to `PEND_LV_SIU_LINK`. Finally, once the message is free to enter the inter-node link, its state changes to `ENTER_LINK`.

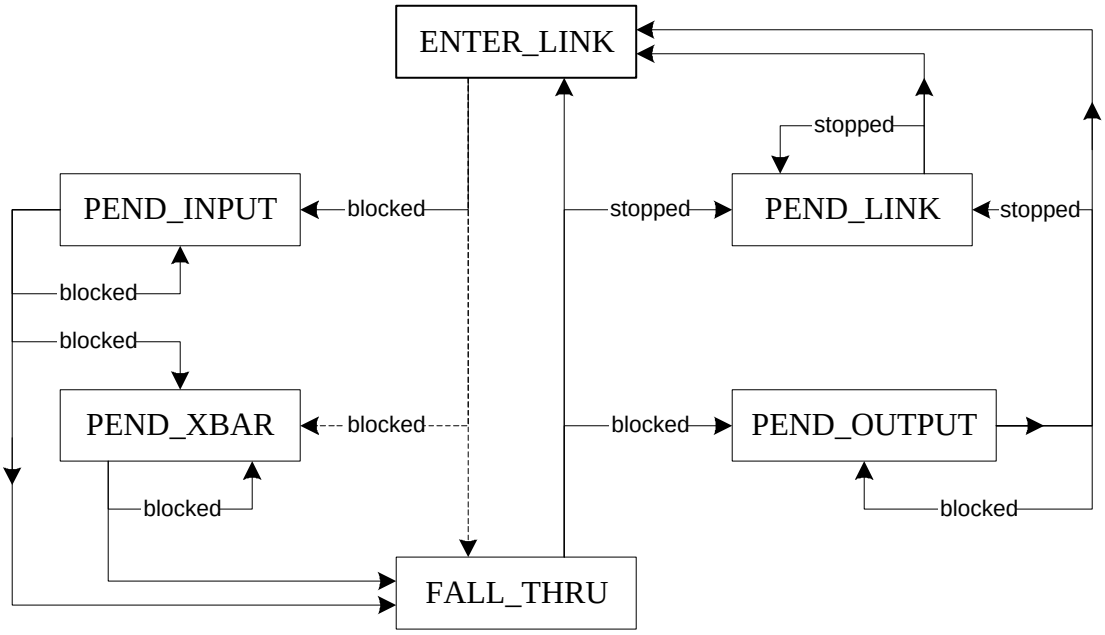


Figure B.5: **States involved in traversing a switch**

The last diagram, in Figure B.5, shows the states involved in traversing a switch. In this case, we both start and stop with the message in an inter-node link (the `ENTER_LINK` state). When the message enters the switch input port, if it is free to traverse the crossbar, it can change its state to `FALL_THRU`. However, if the crossbar is already claimed, it must block on the crossbar and change its state to `PEND_XBAR` until the crossbar is free. If a previous message from the same inter-node link is still exiting the input port, the message state changes to `PEND_INPUT` until it is the first or only message in the input port. At this time it may either be blocked at the crossbar or free to traverse the switch. The transitions from `ENTER_LINK` to both `PEND_XBAR` and `FALL_THRU` are marked specially, because they do not actually occur in the code, even though this is conceptually

what happens. In fact, the state is always changed to PEND_INPUT, from which it may or may not enter the PEND_XBAR or FALL_THRU states. The message stays in the FALL_THRU state until the head flit has traversed the crossbar link. When the head flit enters the crossbar buffer, it may be blocked from entering the link by a previous message, in which case the arriving message changes to the PEND_OUTPUT state. If the message is STOPped (either immediately or after having been blocked) its state changes to PEND_LINK. Finally, once the head flit can enter the inter-node link, its state changes to ENTER_LINK.

B.4 Isotach Level

The Isotach level can be broken into several rough areas, some of which are actually class heirarchies. These areas are the Isotach message heirarchy with CIsoMessage at the base, the Isotach nodes (consisting of the PE, SIU and TM classes), and the group of machines within the SIU which form their own heirarchy with CSiuMachine at the base. Unlike the Myrinet level of the simulator, the Isotach level is not "message-centric". Instead, the Isotach messages are acted upon and within the SIU are moved from one machine to the next each cycle. The classes within the second two heirarchies, which we refer to as the Isotach nodes and the SIU machines, each have DoCycle() and CommitChanges() functions that are called each cycle from the CMyrinetNetwork functions of the same name.

The CIsoMessage class provides the basic functionality common to all Isotach messages such as source information, destination information, and message identification. The subclasses of CIsoMessage provide the specific Isotach message types: CToken for tokens, CSharedRef for shared references (messages that are constrained by Isotach logical time), CNonIso for non-Isotach messages, and CMarker for the marker messages. As there are a number of different types of markers, the CMarker class has its own subclasses: CBarrierMarker for barrier notifications from a PE to its SIU, CEoiMarker for end-of-isochron notifications, and CEopMarker for end-of-pulse notifications from an SIU to its PE. The CEoiMarker is only used internally within the SIU as a means of letting the sender know (from the measurer) that all shared references in an isochron have been received. Generally, the implementation of each of these classes is straightforward, but one note needs to mentioned concerning virtual function calls. Isotach messages are passed from one machine and node to another by moving a CIsoMessage pointer. Although this allows the same mechanism to be used to move all Isotach messages, it does lose the subclass type information. In order to avoid

lots of ugly (and unsafe) typecasting, all functions defined for a derived class are defined as virtual functions in `CIsoMessage` with a default definition that signals an error. Thus, a derived class can "redefine" the function if appropriate, or if the function is inappropriate for that class, it need do nothing.

The SIU is encapsulated within the `CInLinkSiu` class.[3] This class is really not much more than a wrapper for all the SIU machine objects, although it does do initialization including creating the queues that connect each of the machines. There is one machine for each of the units described in the "Design of the Isotach Prototype" working paper, each machine class named appropriately. All of the machines are ultimately derived from the `CSiuMachine` base class. This base class defines a `DoCycle()` function that provides the basic task processing mechanism that keeps track of the correct number of task cycles as defined in the parameter file. Additionally, it automatically checks whether the alternate task time should be used for each new task. The base class is an abstract class; most importantly, it requires subclasses to define a `DoTask()` function and an `IsTaskPending()` function. `DoTask()` is used for executing the actual machine specific duties (once a task has been designated completed), and `IsTaskPending()` is used to define how the machine detects that new processing needs to be done (usually by checking if anything has arrived on an input queue). Since certain configurations of machines occur frequently, several subclasses of `CSiuMachine` have been written to "automate" some of the common functions, but these subclasses are still abstract classes in that they all at least require a `DoTask()` function to be defined. Figure B.6 shows the heirarchy of SIU machine classes. The SIU machines that are gray are unimplemented classes in that they do not perform the task ascribed to it in "Design of the Isotach Prototype". The `CSimpleMachine` class is used for machines that have a single input queue (of Isotach messages); `IsTaskPending()` is implemented to check whether anything has arrived on that queue. The `CSimpleFilterMachine` is identical to the `CSimpleMachine` except that the filter machines' input queues contain Myrinet messages instead of Isotach messages. Finally, the `CMultiMachine` class is used for an arbitrary number of input and output queues (of Isotach messages). The `IsTaskPending()` function is defined to round-robin through the input queues. Two machines, the `CToNetMerger` and the `CToHostMerger` have some additional complexities as described later.

The PE is encapsulated within the `CProcElem` class. This class has a very simple task duty. Every time `DoTask()` is executed it first processes all incoming messages. Then if a barrier message

[3]The name goes back to when there were actually a couple different design ideas concerning the placement of the SIU.

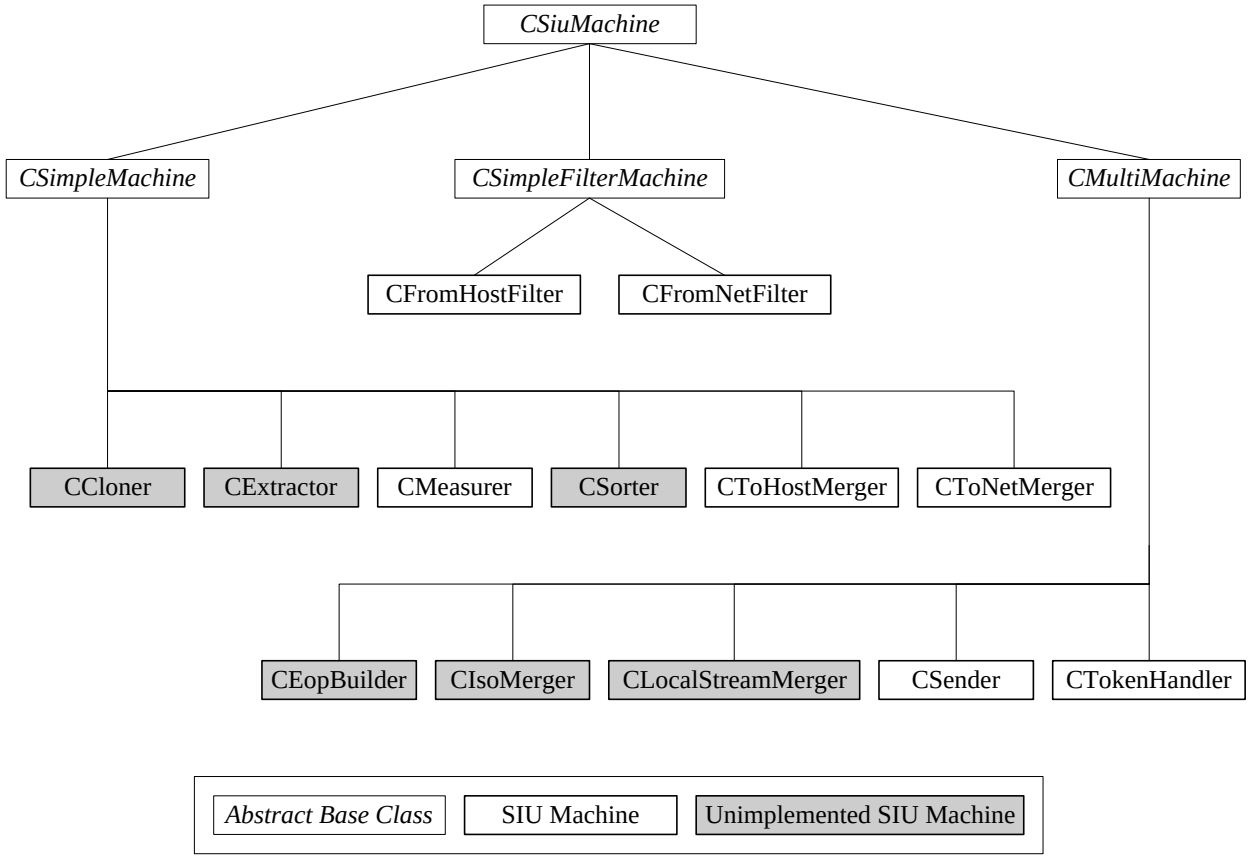


Figure B.6: **The SIU machine heirarchy**

`CSiuMachine` is the root abstract base class for all the SIU machines. The other abstract base classes provide "automated" functionality for detecting when a task needs to be performed. The SIU machines that are gray are not implemented, and the other SIU machines are mostly if not totally implemented.

(either Isotach or non-Isotach) needs to be generated that is its task. If no barrier message needs to be generated, then some other type of message (or no message at all) is generated based on the distributions and probabilities set in the parameter files.

The TM, which is encapsulated within the `CTokenManager` class, is somewhat more complicated. In fact, a separate class, the `CTmTokenHandler` has been created to specifically implement the TM's token and barrier algorithm. It is this class that keeps track of which tokens have arrived on which inputs, which of these tokens are barriers, and what the current state of an outgoing wave might be.

Finally, the `CProcElem`, `CTokenManager`, `CToNetMerger`, and `CToHostMerger` classes also have some operations at the Myrinet level. All of these classes are derived from the abstract base class

CMyrinetNode[4], which defines an array of input and output ports (the size of which is determined by the subclass). Each of these classes initializes Myrinet ports and is capable of generating, sending, and receiving Myrinet level messages. Note that the other class derived from CMyrinetNode is CMyrinetSwitch, which exists wholely at the Myrinet level and is discussed in detail in Section 3.3.

B.5 Data Gathering and Statistics

The simulator interacts with the data gathering mechanism through one object, the global observer of type CObserver. The interactions are primarily in the form of message departure and arrival notifications, but additionally the main loop (in doSim.C) notifies the CObserver of the start and end of cycles, sections, and runs. During initialization, the CObserver also gets pointers to all the various simulator elements including nodes and ports. It is the CObserver class that compiles all the statistical data and provides the printing mechanism.

CObserver maintains the statistics for tokens and barriers by using the classes CTokenStat and CBarrierStat respectively. Message notifications and start/end notifications received by CObserver are passed on to each of these objects. CTokenStat, through the use of three other classes, CTokenLatencyStat, CPulseLengthStat, and CTokenBarrierRatioStat keeps track of the token link latencies, the pulse lengths and the ratios of barrier tokens to regular tokens. CBarrierStat keeps track of epoch lengths through the use of the CEpochLengthStat class. A diagram of these class dependencies is given in Figure B.7.

The fundamental statistics classes consist of the general stat classes CStat and CCumStat, and the histogram heirarchy composed of base class CHistogram and subclasses CIntHistogram and CRealHistogram. Floating-point values can be entered into CStat, which will maintain a running average, standard deviation and count of the number of entries. This is an efficient class for keeping track of basic statistical information if no entry is history is required. CCumStat is similar to CStat, except that it takes two values and keeps track of the data on each value separately (by using the CStat class no less). This class is primarily used to make cumulative data gathering easier in that the two values are usually an average and a standard deviation. Thus one can easily determine the average and standard deviations of a number of data points (which each consist of an average and

[4]Yes, this means that CToNetMerger and CToHostMerger are multiply derived classes.

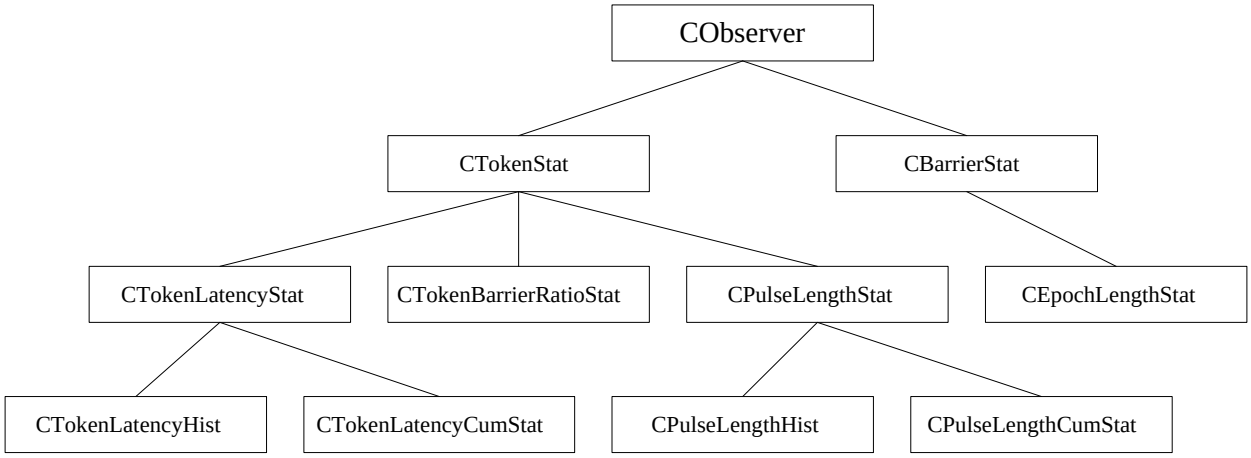


Figure B.7: **Dependencies of statistics classes**

`CObserver`, at the top of the dependency heirarchy, is the class through which the rest of the simulator interfaces with the statistics mechanism. This class relies on two classes, `CTokenStat` and `CBarrierStat`, to handle the statistical details for tokens and barriers respectively. These two classes in turn rely on several other classes to handle further statistical details.

a standard deviation). See Appendix A.6 to see why this is useful.

The histogram classes, like the general stat classes, provide averages and standard deviations, but additionally it keeps track of how many values occur within defined ranges. These ranges, called buckets, can be set to any arbitrary size, but default to 1. Note that if the bucket size is larger than 1, the averages and standard deviations might be less accurate, as these statistics are computed using the mid-point of each range. These classes naturally consume more memory and are a little slower than the stat classes.

Finally, there are four other simple classes used to process statistics. `CTokenLatencyHist` and `CTokenLatencyCumStat` are helper classes used by `CTokenLatencyStat`, and `CPulseLengthHist`, and `CPulseLengthCumStat` are used by `CPulseLengthStat`. These assist in keeping track of arrays of histograms and `CCumStat` objects. Each array entry corresponds to a statistic for a pair of nodes within the network.

B.6 Coding Standard

As this simulator has grown to quite respectable proportions, we have developed a simple coding standard (which has been kept to pretty well) to help with code management. This coding standard

consists mostly of naming conventions, but there are a couple other tips.

- Constants are written in all capital letters. If the constant consists of more than one word, each word is separated by an underscore. (e. g. `NUM_TM_PORTS`). This naming convention applies to both constants defined with the `const` keyword and enumerated type values.
- User defined types are written in lower-case, but the first letter is capitalized. If the type consists of more than one word, each word is indicated by capitalization. Furthermore, all user-defined types must in a "`T`" (e. g. `BoolT` and `StatTypeT`). This naming convention applies to both typedefs and enumerated types.
- Global functions are written in lower-case. If the function name consists of several words, each word after the first is indicated by capitalization (e. g. `latencyTypeToStr(...)` and `sqrRootDistr(...)`).
- Class names begin with a "`C`" and are written in lower-case. All words are indicated by capitalization (e. g. `CIsoMessage`).
- Template names begin with a "`T`" and are written in lower-case. All words are indicated by capitalizations (e. g. `TPortQueueElem`).
- Class methods (functions) are written in lower-case, but each word, including the first, is capitalized (e. g. `IsReady(...)` and `RegisterPe(...)`).
- Variables are written in lower-case. If the variable consists of more than one word, each word after the first is capitalized. (e. g. `isDone` and `orderCounter`).
- For all naming conventions that specify lower-case, words that normally consist of all caps (like SIU) are treated like normal words. (e. g. `CInLinkSiu` and `siuGeneratesToken`).
- Certain types of variables are prefixed with letters to indicate their type.

  `p` — pointer
  `r` — reference (generally only if not a const reference)
  `a` — array (LEDA array data type preferably, but also C++ arrays)
  `l` — list (LEDA list data type)
  `q` — queue(LEDA queue data type)

`s` — set (LEDA set data type)

`m` — map (LEDA map, h_array, dictionary, sortseq, etc. data types)

`t` — table (LEDA array2, node_matrix, etc. data types)

`g` — graph (LEDA graph, ugraph data types)

If several prefixes are used, then they are read left to right. For example `lpIsoMsg` would be a list of pointers to IsoMsgs, whereas `plIsoMsg` would be a pointer to a list of IsoMsgs.

- Variables within the scope of a class (i. e. member data) are additionally prefixed with an "`m_`" as in `m_lpIsoMsg`.
- Globally scoped variables are additionally prefixed with a "`g_`" as in `g_observer`.
- In general, inquiry member functions tend to start with "`Is`", safe upcasting member functions start with "`As`", and accessor member functions start with "`Get`" or "`Set`".
- Member data should usually be declared private. Derived classes should access the data through protected accessor functions. Following this convention helps improve encapsulation, as changing the structure of data in a base class does not require one to examine and possibly modify all derived classes. (This convention also improves certain scope issues.)
- A file should contain code for only one class at a time (header or source), and the file should have the same name as the class. Additionally, header files should end in "`.H`", source code files should end in "`.C`", and inline source code files (if the inline code is not included in the header file) should end in "`.IC`".
- The files `template.H` and `template.C` can be used to conveniently create and organize a new class. This is recommended, because it can make it easier to locate certain portions of a class.
- Although not part of a coding standard per se, if a LEDA container class is used to hold a user-defined class, then the iostream operators << and >> should be defined to avoid a slough of compilation warnings. These function definitions can just be stubs — examples can be found in `CNodeInfo.H` among other files.

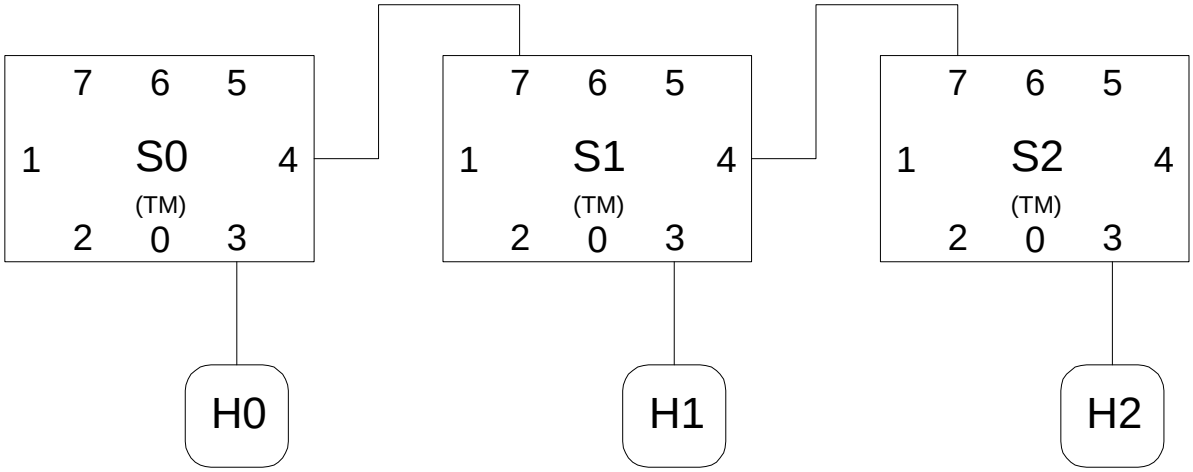
Appendix C

Network Topologies

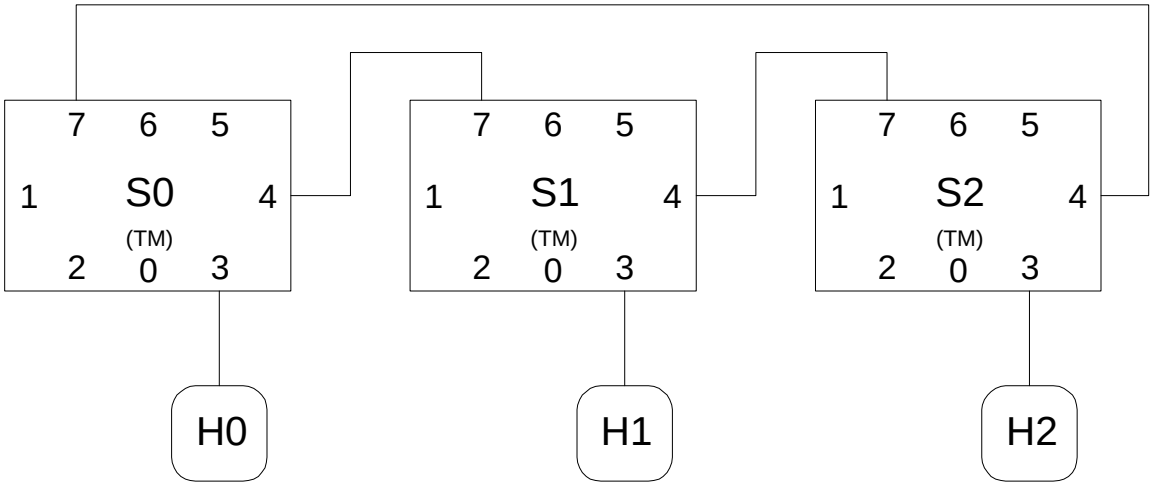
This appendix contains diagrams of all the pre-defined network topologies that are included in the code release. The network data files are located in the `topologies` subdirectory. Each file is named `network-␣␣␣␣.dat` where the blanks are filled with the topology name (e. g. 3rb1 or 6rb5). Note that the selected file will have to be renamed to `network.dat` to be used with the simulator.

The format of the topology names is a number followed by two letters followed by another number. The first number indicates how many switches are in the topology. The first letter (second character) indicates the configuration of the switches. A "`b`" indicates a "bus" topology where the switches are all connected in a single line. An "`r`" indicates a "ring" topology, where the switches are connected in a single line additionally the first and last switches are connected. An "`m`" indicates a "mesh" topology, where the switches are organized into a grid. The second letter (third character) indicates how the SIUs are distributed. Currently the only choice is "`b`", which indicates "balanced" or all switches are connected to the same number of SIUs. The second number (fourth character) indicates how many SIUs are connected to each switch.

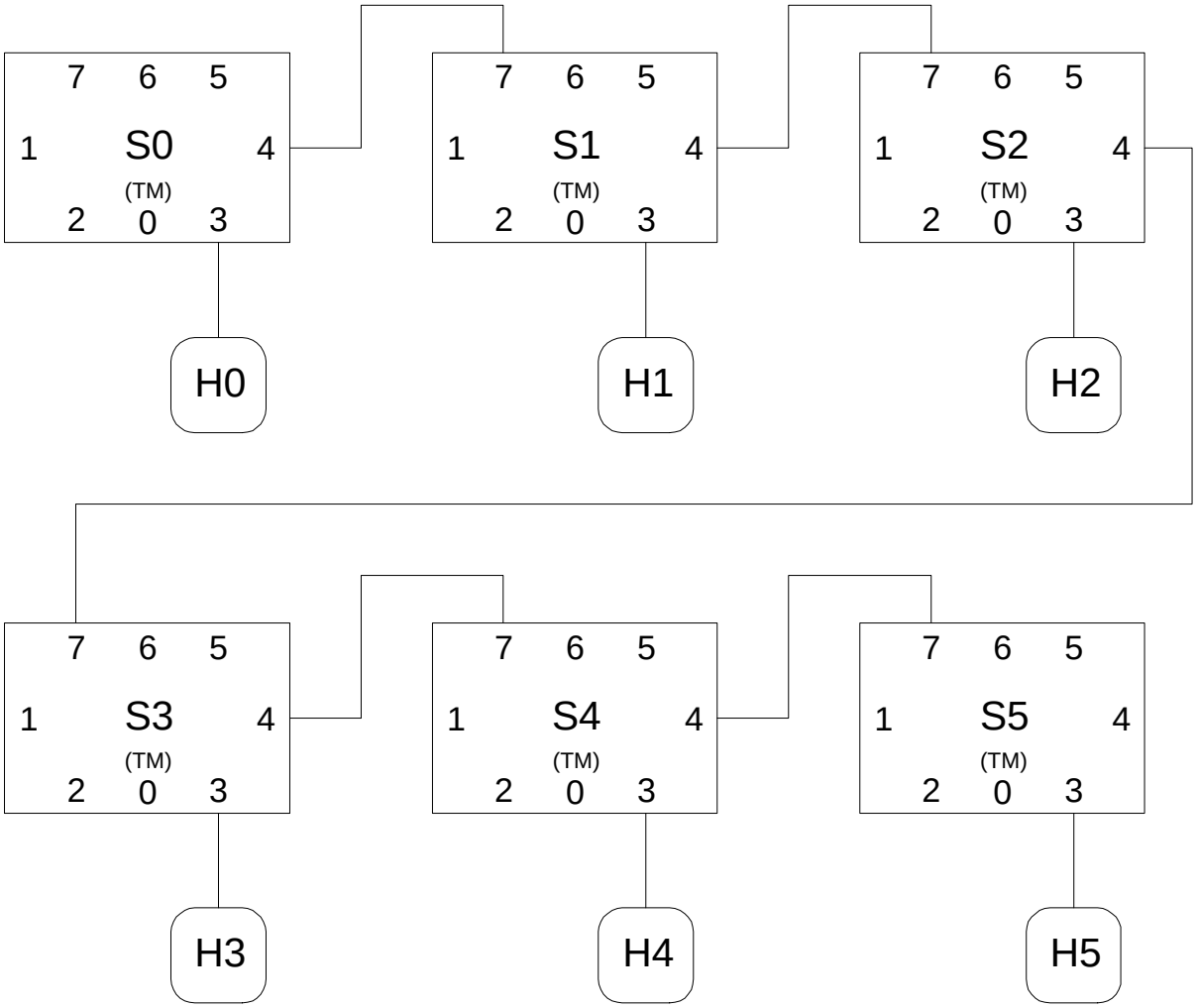
Network 3bb1



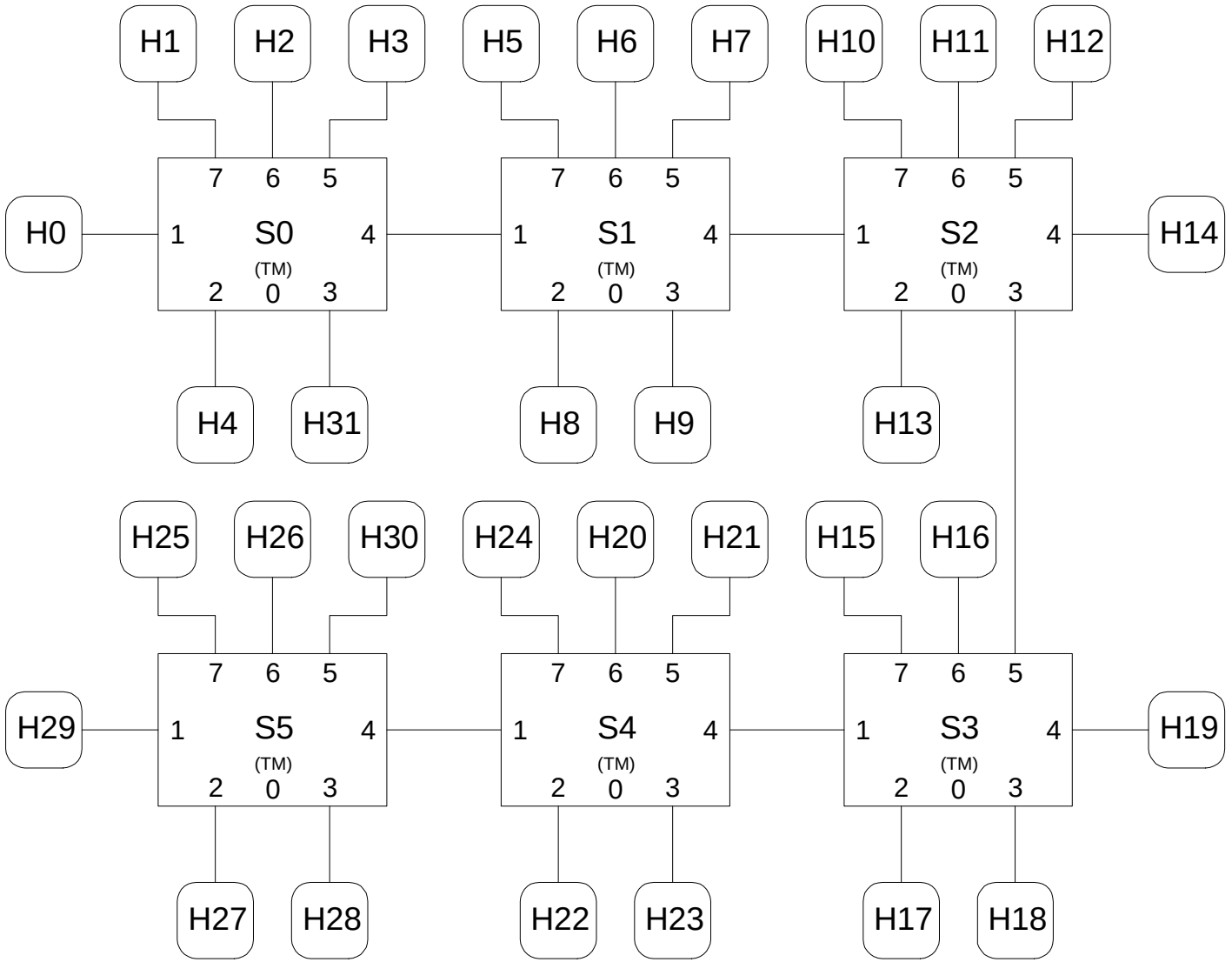
Network 3rb1



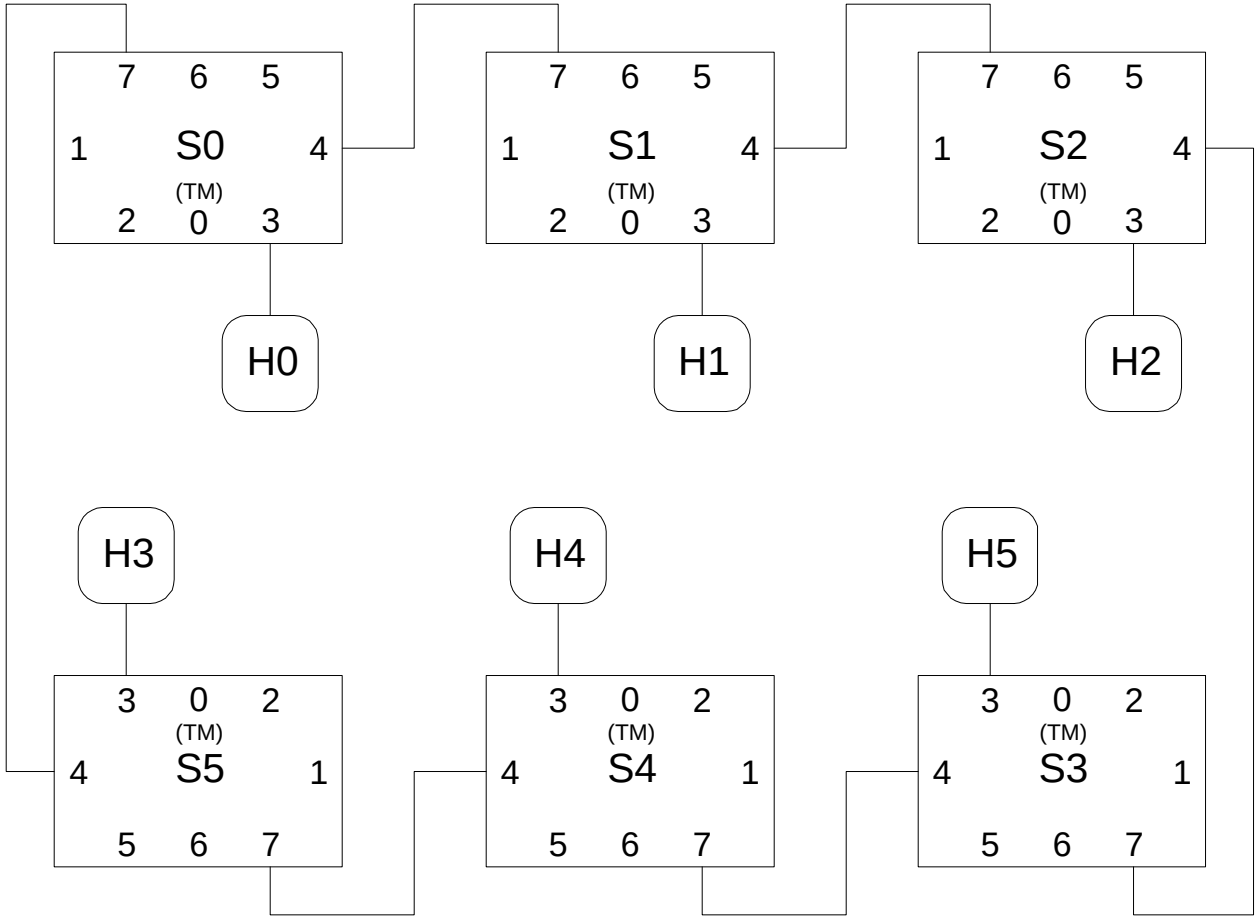
Network 6bb1



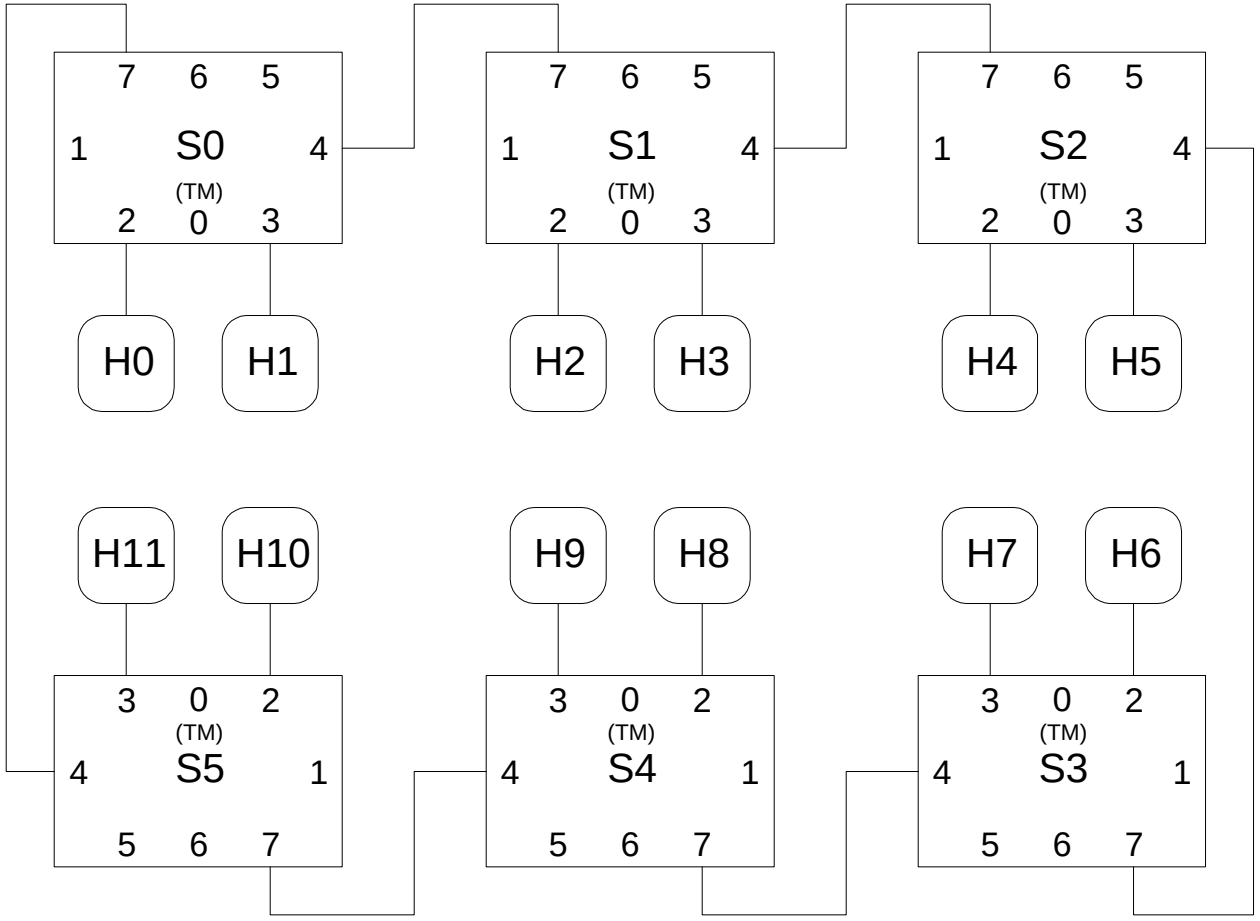
Network 6bb5+1e (1 on ends)



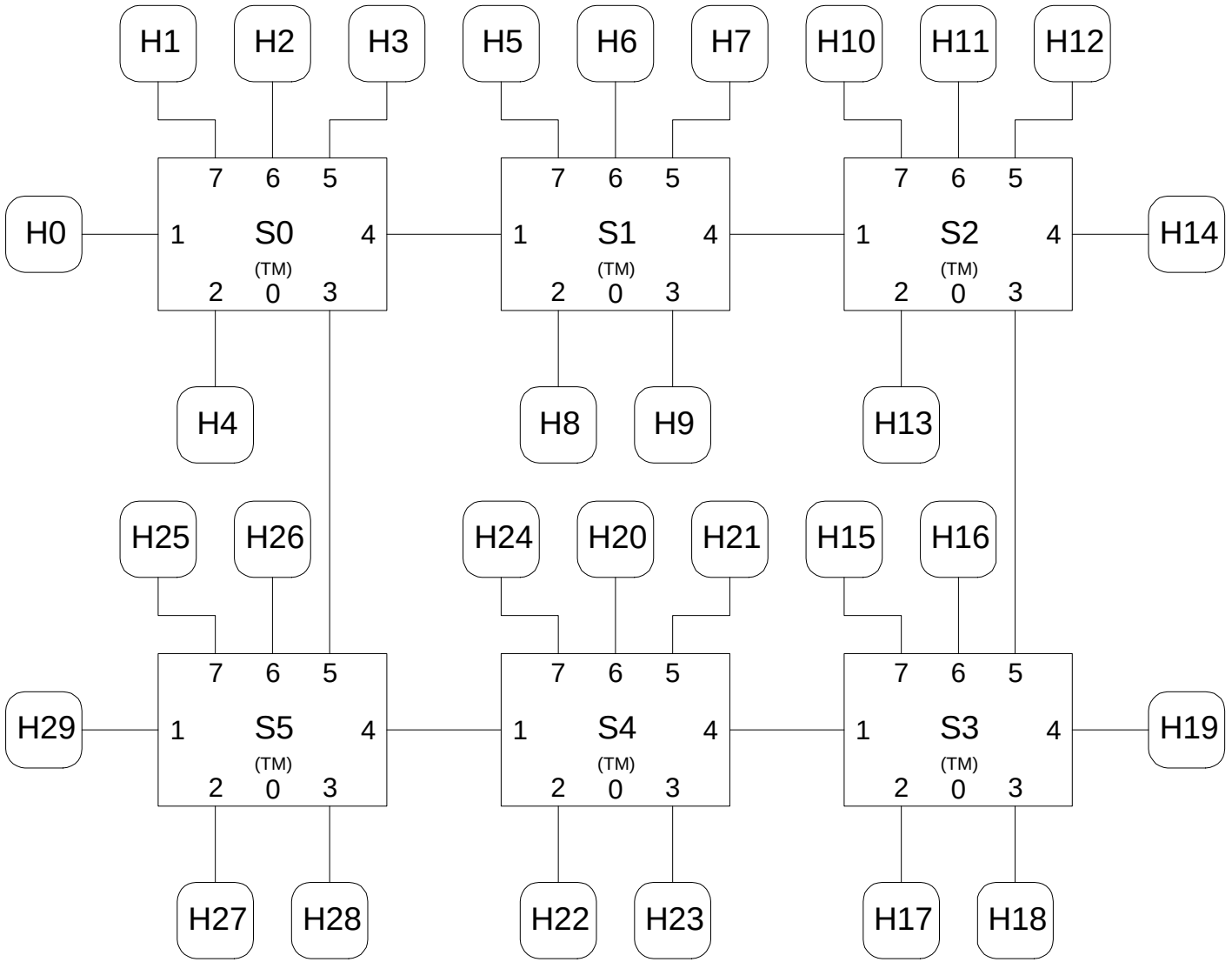
Network 6rb1



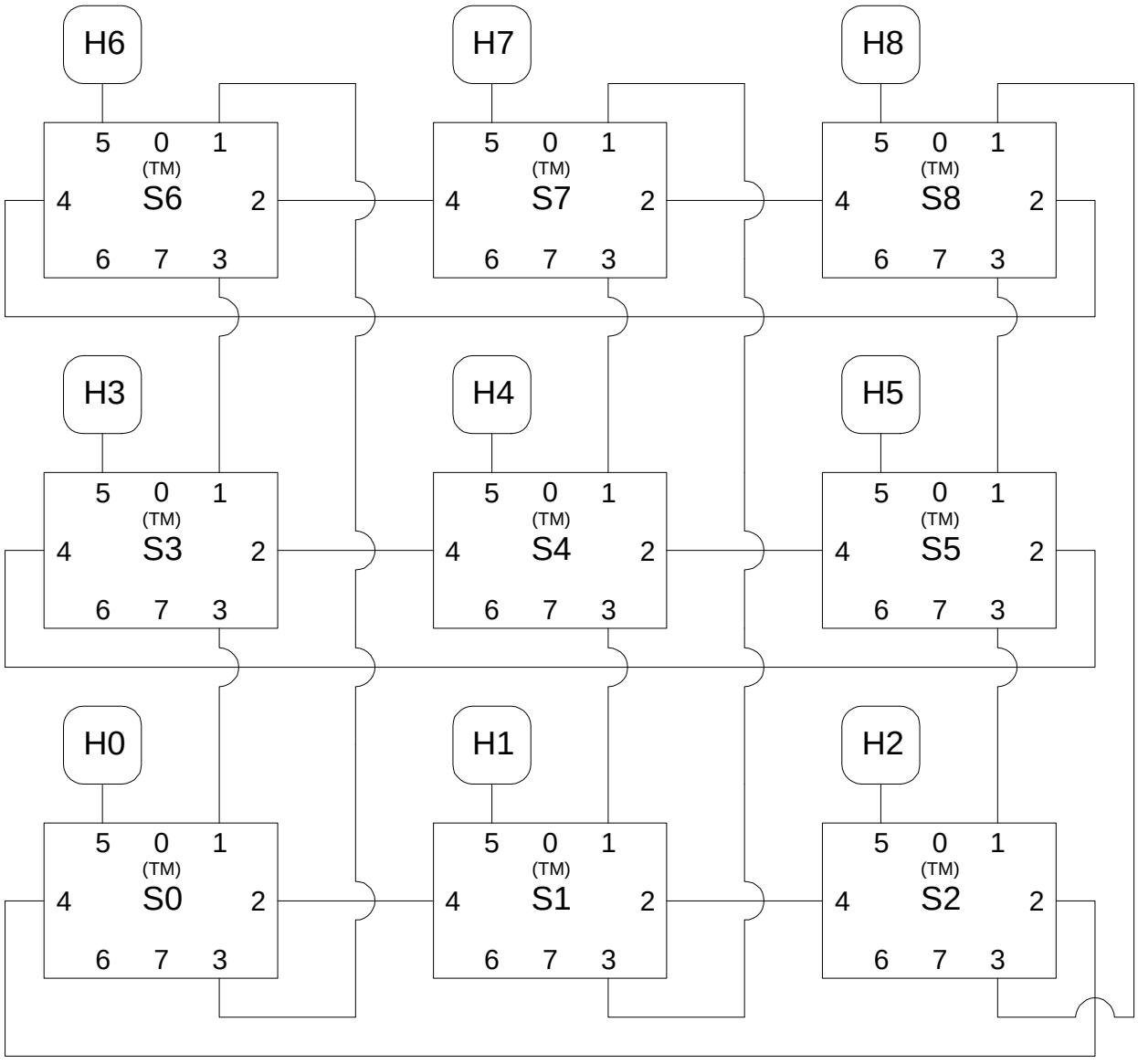
Network 6rb2



Network 6rb5



Network 9mb1



Bibliography

[1] Avalanche scalable parallel processor project. Avalanche project information is available from http://www.cs.utah.edu/projects/avalanche/avalanche-publications.html.

[2] Yitzhak Birk, Phillip B. Gibbons, Jorge L. C. Sanz, and Danny Soroker. A simple mechanism for efficient barrier synchronization in mimd machines. Research Report RJ 7078 (674141), IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, October 1989.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet — a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995. This article is also available from http://www.myri.com/research/publications/Hot.ps.

[4] Frank Brill. Myrinet simulation programmer's guide. Available in pre-Isotach simulator code release, 1996.

[5] Frank Brill. Myrinet simulation user's guide. Available in pre-Isotach simulator code release, 1996.

[6] Eugene D. Brooks III. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[7] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier syncronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[8] Rahmat S. Hyder and David A. Wood. Synchronization hardware for networks of workstations: Performance vs. cost. Technical report, Computer Sciences Department, University of Wisconsin—Madison, 1210 West Dayton Street, Madison, WI 53706, n.d.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] Kurt Mehlhorn and Stefan Näher. The LEDA platform of combinatorial and geometric computing. Chapter excerpt available from http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html, 1997.

[11] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, pages 62–76, February 1993.

[12] Dhabaleswar K. Panda. Fast barrier synchronization in wormhole k-ary n-cube networks with multidestination worms. *Journal of Future Computer Systems*, November 1995.

[13] John Regehr. An Isotach implementation for Myrinet. Technical Report CS-97-12, Department of Computer Science, University of Virginia, Charlottesville, VA 22903, May 1997. This report is also available from http://www.cs.virginia.edu/~isotach/pubs.html.

[14] Paul F. Reynolds, Craig Williams, and Raymond R. Wagner, Jr. Isotach networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(4), April 1997. This article is also available from http://www.cs.virginia.edu/~isotach/pubs.html.

[15] Craig Williams. *Concurrency Control in Asynchronous Computations.* PhD thesis, Department of Computer Science, University of Virginia, Charlottesville, VA 22903, January 1993. This dissertation is also available from http://www.cs.virginia.edu/~isotach/pubs.html.