INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313 761-4700 800 521-0600

Order Number 9208576

Networking in distributed real-time systems

Kandlur, Dilip Dinkar, Ph.D.

The University of Michigan, 1991

U·M·I

300 N. Zeeb Rd.
Ann Arbor, MI 48106

NETWORKING IN DISTRIBUTED REAL-TIME SYSTEMS

by

Dilip Dinkar Kandlur

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1991

Doctoral Committee:

Professor Kang G. Shin, Chairman
Associate Professor John R. Birge
Assistant Professor Chinya V. Ravishankar
Assistant Professor Stuart Sechrest
Associate Professor Toby J. Teorey

To my parents and Sharmila

ACKNOWLEDGEMENTS

It is impossible to individually acknowledge all the people who have contributed, both directly and indirectly, to this dissertation. However, I would be amiss if I did not specially thank the people who have influenced me the most.

My deepest gratitude is to my advisor Professor Kang G. Shin for his constant encouragement and support throughout the course of this work. In spite of all the demands on his time, he has always been eager to discuss our problems and his insightful comments and suggestions have been invaluable. Moreover, it would be hard to find a person who is more concerned about his students. Similarly, I would like to thank Professors Toby Teorey and Chinya Ravishankar for their guidance during the initial stages of my graduate program, and for serving on this doctoral committee. I would also like to express my appreciation to the other members of this committee, Professors Stuart Sechrest and John Birge for their constructive criticisms on this dissertation.

I have benefited greatly from my discussions with several past and present members of the Real-time Computing Laboratory. In particular, Parmeswaran Ramanathan helped me formulate the ideas on reliable broadcasting and traffic routing. James Dolter helped me understand the operations of the HARTS routing and packet controllers. Parmesh and Jim have also provided assistance in various other things, too numerous to list here. Daniel Kiskis helped me develop the preliminary version of the HARTOS operating system.

I would like to gratefully acknowledge the Department of EECS, the Office of Naval Research, and IBM Corporation for providing financial support during the course of my graduate program. Thanks to Brian Aupperle for developing the Rackham thesis style which I have used to format this document. Thanks also to Sriram Padmanabhan, Padmanabhan Krishnan, and Krishna Reddy for making my stay in Ann Arbor more enjoyable.

Finally, I thank my wife Sharmila for her constant support, her patience and understanding.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Background

A commonly used definition of a real-time system is that it is one in which the value of a computation depends not only on the logical correctness of the results, but also on the time at which the results are produced [SR88]. This definition reinforces the notion that time is one of the most important entities in the system, and there are timing constraints associated with system tasks. While this definition is broad, and applies to several classes of systems like multimedia communication systems, on-line transaction processing systems, and process control systems, our focus is restricted to real-time process control systems. Examples include control systems for aircraft, spacecraft, nuclear power plants, and drive-by-wire automobiles.

In addition to timing constraints, another important characteristic of real-time control systems is their stringent reliability requirement, since a failure of the control system can lead to a disaster. The specification of the reliability depends upon the application. For example, the reliability requirements for commercial transport aircraft and spacecraft are specified in terms of the allowable probability of failure per mission, and a figure of 10^{-9} has been specified by the NASA for commercial aircraft for a 10–hour flight [G+84]. In order to achieve these stringent requirements the system must have the ability to withstand component failures, so fault-tolerance plays an important role in the design of these systems.

Distributed processing systems are potentially well suited to meet the performance and reliability requirements of real-time control systems. These systems offer several advantages such as parallel computation, graceful performance growth, and graceful degradation in the presence of faults. Moreover, a distributed system is ideally suited for environments which have considerable physical separation between the components to be controlled. The availability of inexpensive and powerful microprocessors also makes these systems cost-effective.

In the design of distributed systems for real-time applications, along with processors and memory components, the interconnection network is an important component. One of the goals in the design of the interconnection network is to provide reliable communication in the presence of component failures. The approach taken in multicomputers like SIFT [G+84] and MAFT [KWFT88] is to provide a fully-connected network, where each node is connected to every other node with dedicated point-to-point links. Although this method is extremely reliable, it does not scale and can be used only in systems with a small number of nodes. As a case in point, both SIFT and MAFT have less than 10 nodes.

For larger systems, it is necessary to use networks which are not fully connected. A notable approach to providing fault-tolerant communications is the AIPS *virtual bus* scheme [Lal87, LHA91]. The AIPS network controllers are connected by multiple point-to-point links, but they are configured under software control to act as a single virtual bus. However, there is a significant delay incurred in traversing through the repeater stages in a controller node, so the end-to-end latency is substantial when multiple links are to be traversed. This results in long bit-holding times for the contention resolution protocol used to gain access to the virtual bus, which can adversely affect the throughput. Also, the virtual bus configuration does not permit simultaneous communication over disjoint paths in the network.

In contrast to these approaches, we consider a (partially connected) point-to-point interconnection network with a regular structure as a good candidate for use in real-time control systems. Examples of such networks include hypercubes and meshes [Sei85, CSK90]. The existence of multiple disjoint paths between nodes in these networks make them robust to link and node failures. Also, the links in this network operate in parallel and this results in a higher total throughput than that of shared medium networks like buses and rings.

Typically, these networks have been operated in a *packet switching* mode. In this mode, if a message has to traverse several network nodes to reach its destination, packets are stored at each intermediate node and then transmitted forward. One major drawback of packet switching is the large delay in communication which arises from the store and forward mechanism. Another possible mode of operation is *circuit switching*, in which the route to the destination is reserved before the message is transmitted. Since the route is reserved, the packet can be directly transmitted from the source to the destination without any buffering delays. The problem with this method is the overhead associated with the circuit set-up, which makes it unsuitable for short messages. The circuit set-up overhead also results in a wastage of network bandwidth and low utilization.

The *virtual cut-through* switching technique can provide low-latency communication in partially connected point–to–point networks. This technique, first proposed by Kermani and Kleinrock [KK79], has flavors of both circuit and packet switching. In this scheme, messages arriving at an intermediate node do not always get buffered, instead, they are forwarded to the next node in the route if a circuit can be established. This differs from conventional packet-switching schemes in the sense that messages do not always get buffered at an intermediate node. It also differs from circuit switching schemes since messages do not wait for the entire circuit to the destination to be established before proceeding along the route.

Advances in VLSI technology have now made it possible to implement sophisticated switching schemes, like virtual cut-through, which reduce the message delivery time significantly. Network controllers like the Torus routing chip [DS86, DS87] and the HARTS routing controller [DRS89] are notable examples. It is therefore feasible to use a multicomputer based on a point–to–point network with virtual cut-through switching for many real-time applications. Many of the issues considered in this dissertation deal with communications in a multicomputer system of this kind.

1.2 Research Objectives

Although distributed systems offer many advantages for real-time applications, they also present several challenging problems. This includes providing guarantees for timely delivery of messages and managing the network to realize its potential for high performance and fault-tolerance. The main focus of this dissertation is to address problems related to providing time-constrained and fault-tolerant communication in distributed real-time systems.

In any real-time system, a significant number of tasks have time constraints associated with them. Communication between these tasks has to be *predictable* and time-constrained because unpredictable delays in message delivery could affect their execution. For example, sensor data has to be delivered to the processing task within certain time bounds, so that the task can produce its control output in time. In order to make inter-task communication predictable, the sending task must be given a guarantee on the delivery time of the message. Moreover, this delivery time should not be an artifact of the network, but should be determined by the needs of the sending task. The problem of providing timing guarantees for inter-task communication is especially difficult in distributed systems, because network delays and characteristics have to be included in the analysis. Our efforts

are aimed at developing a scheme which gives *a priori* guarantees on message delivery for inter-node communication with delivery time constraints.

In the type of network that we consider, there are several different paths between any given pair of nodes. This poses the problem of selecting routes for inter-task communication, and our aim is to select routes which preserve the benefits of virtual cut-through switching. In this switching technique, since messages do not necessarily get buffered at intermediate nodes, the delays encountered are smaller than those for a packet switching scheme. The automatic forwarding also means that if a cut-through switch is established, the packet is not seen by the processor at that intermediate node. Hence, the load imposed on the network processors at the intermediate nodes is smaller than that for a packet switching scheme. However, the communication latency is critically dependent upon the number of times that a packet gets buffered at intermediate nodes. The emphasis of our traffic routing scheme is on choosing routes so that the congestion in the network is avoided, and the number of bufferings is minimized. This work also yields an algorithm which can be applied to select routes for traffic with timing constraints.

In a real-time system, it is essential to provide a common time-base for all the system nodes. The time base is required so that common deadlines can be specified and tasks on different nodes can coordinate and complete their operations before the specified deadline. In order to establish a common time-base, it is necessary to synchronize the local clocks on the individual nodes in the system. Clock synchronization can be achieved by using either hardware or software solutions. Hardware solutions can achieve very tight synchronization, but they do so at the expense of employing additional lines for clock signals or using a separate clock network. Software solutions, on the other hand, use messages to periodically exchange clock values between nodes and they adjust the individual clocks based on these clock values. In [RKS90], we developed a synchronization scheme which strikes a balance between the additional hardware required and the clock skews that can be attained. It is a software algorithm that requires no modifications to the network, and uses timestamping hardware at the nodes to achieve reasonably tight synchronization.

This synchronization scheme relies on a mechanism for broadcasting clock messages reliably from any node in the system. Broadcasting is also an important operation in many other algorithms like distributed diagnosis, resource lookup, etc. An obvious shortcoming of a point–to–point interconnection network is that broadcasting is expensive and non-trivial. We study the problem of providing support for efficient broadcasting in a point-to-point network with virtual cut-through switching. Furthermore, we also address the problem of

reliable broadcasting in the presence of unknown (or even Byzantine) faults, in a mesh network.

In addition to developing analytical solutions for these problems, our aim is to implement them to provide communication services for applications. Although communications support is critical for a distributed system, it is a system overhead from the perspective of applications. If the overhead is high, it can interfere with execution of application tasks. Thus, our objective is to explore the use of a dedicated communication coprocessor to provide these services. A dedicated communication coprocessor may add to the cost of the system, but it can be used to perform additional functions related to monitoring and system diagnosis.

In summary, the objectives of this dissertation are to:

- develop a scheme which supports time-constrained communication
- improve the performance of virtual cut-through switching using traffic routing
- provide a fault-tolerant broadcasting mechanism for mesh networks
- integrate these services into the operating system and exploit the use of a communication coprocessor to provide these services.

1.3 Approach

The goal of this dissertation is to study and solve problems related to communication in distributed real-time systems. Our endeavor is to develop solutions for these problems in a general setting, which would be applicable to a variety of real-time systems. However, we are also interested in implementing our solutions so as to demonstrate their validity and thus develop a base for experimenting with real-time applications. The following subsection gives an overview of HARTS[1], which serves as the implementation vehicle for our research.

1.3.1 HARTS

HARTS is an experimental distributed real-time system, which is being developed in the Real-time Computing Laboratory, The University of Michigan [Shi91]. It is comprised of several multi-processor nodes connected by a point–to–point interconnection network. A

[1] Hexagonal Architecture for Real-Time Systems

HARTS node consists of several Application Processors (APs) which are used for running application tasks, and a Network Processor (NP). The NP contains the interface to the network, buffer memory, and a general-purpose processor. This processor can be used to handle most of the processing related to communication. The network uses the virtual cut-through switching scheme, described earlier in Section 1.1.

The HARTS network has a *C-wrapped hexagonal mesh* topology [CSK90, Ste86], which is a regular, homogeneous graph where each node has six neighbors. It can be defined succinctly as follows.

Definition 1 *A C-wrapped hexagonal mesh of size n is comprised of* $N = 3n(n-1)+1$ *nodes, labeled from* 0 *to* $N-1$, *such that each node s has six neighbors* $[s+1]_N$, $[s+3n-1]_N$, $[s+3n-2]_N$, $[s+3n(n-1)]_N$, $[s+3n^2-6n+2]_N$, *and* $[s+3n^2-6n+3]_N$, *where* $[a]_b$ *denotes a* mod *b.*

The graph can be visualized as a simple hexagonal mesh with wrap links added to the nodes on the periphery. A simple hexagonal mesh looks like a set of concentric hexagons with a central node, where each hexagon has one more node on each edge than the one immediately inside of it. Figure 1.1(a) shows a simple hexagonal mesh of dimension 3, while Figure 1.1(b) illustrates the wrapping scheme for the nodes. An analysis of some of the topological properties of the HARTS network, and its comparison with other topologies, can be found in [CSK90].

The hexagonal mesh offers better connectivity, and thus better fault-tolerance, than a rectangular mesh. Moreover, routing algorithms have been developed [OS89] which exploit the fault-tolerance properties of the network. Compared to the hypercube topology, the hexagonal mesh has the advantages of better scalability and fixed node degree. Also, for small systems (less than 100 nodes), it has better connectivity than a hypercube. The version of HARTS under construction has a hexagonal mesh of dimension 3 and contains 19 nodes.

In the current configuration, the nodes of the HARTS system are VME-bus based Ironics Performer-32 systems (see Figure 1.2). Each node has 1–3 AP cards, a System Controller card, and an Ethernet processor card. The processor cards have a Motorola 68020 32-bit processor, optional memory management unit, and 1 or 4 Mbytes of dual ported RAM which can be accessed from the VME-bus. These cards also have a hardware mailbox interrupt mechanism which generates a CPU interrupt on a write access to the top 256 bytes of the dual-port memory. The System Controller Card arbitrates the VME bus and also contains two serial ports, a SCSI bus controller, and a clock/calendar. The Ethernet

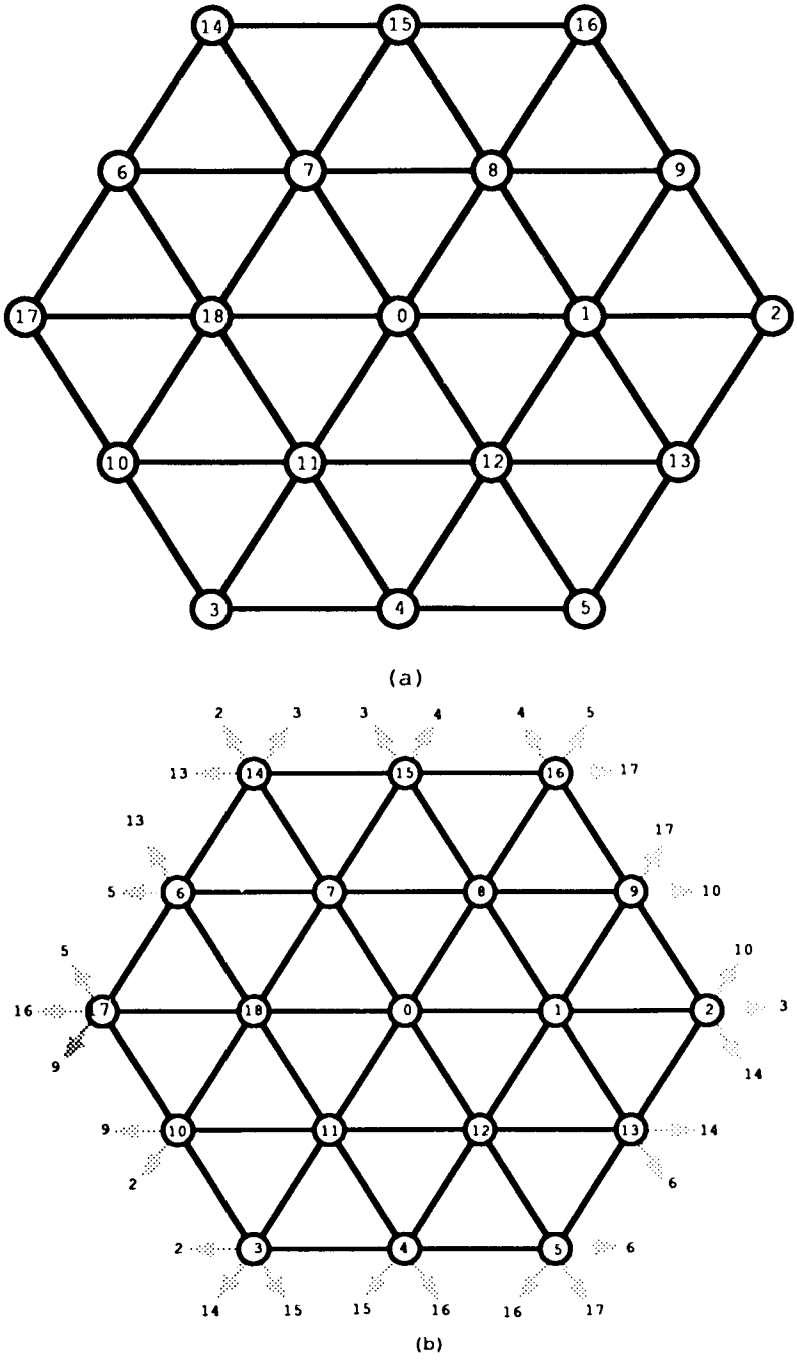

Figure 1.1: A hexagonal mesh of dimension 3 (E-3).

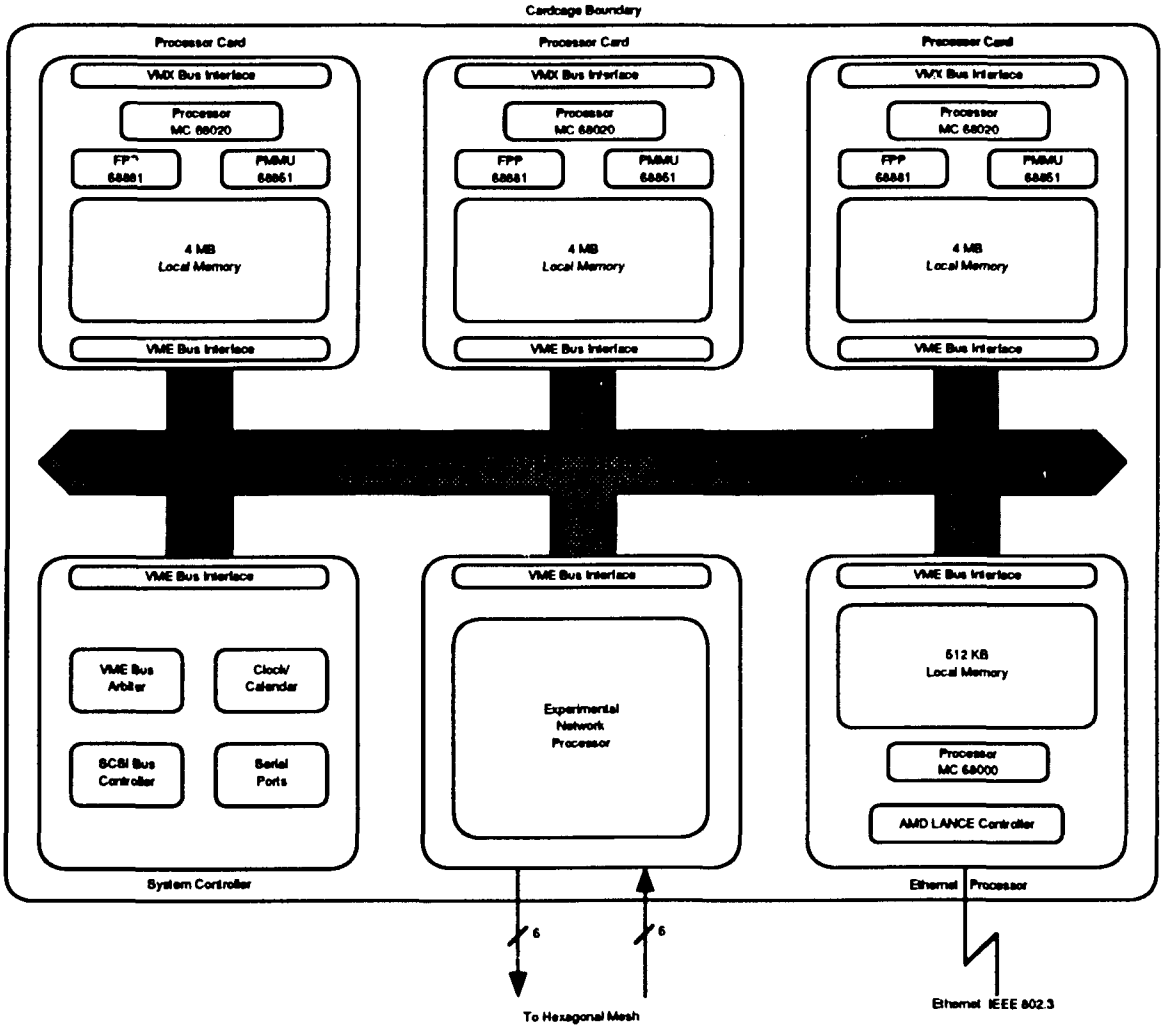

Figure 1.2: Block diagram of a HARTS node.

Processor card ENP-10 uses a 10 MHz 68000 processor, an AMD Ethernet Controller device (LANCE), and provides 512 Kbytes of buffer memory which is also accessible from the VME-bus. Note that the Ethernet is not a part of the HARTS architecture, but it currently serves as the system interconnect while the custom hexagonal mesh network is under development. The Ethernet also serves as a link to the workstations used for software development. A custom routing controller chip [DRS89] has been developed which supports virtual cut-through switching. It implements the data link layer and portions of the network layer of the OSI seven layer model for network communications. This chip will be used in the network processor card and will serve as the front-end interface to the hexagonal mesh.

1.4 Outline of the Dissertation

The structure of this dissertation is as follows. In Chapter 2 we present an overview of our approach to the problem of providing support for communications in distributed real-time systems. We also describe the communication requirements of real-time systems, and propose a scheme for providing predictable inter-process communication in real-time systems which gives guarantees on the maximum delivery time for messages. This scheme is based on the concept of a real-time channel, a unidirectional connection between source and destination. A real-time channel has parameters which describe the performance requirements of the source–destination communication, e.g., from a sensor station to a control site. This chapter provides the framework for the work described in subsequent chapters.

Chapter 3 deals with the problem of establishing real-time channels. Once such a channel is established, the communications subsystem guarantees that these performance requirements will be met. In this chapter, we concentrate on methods to compute guarantees for the delivery time of messages belonging to real-time channels. We also address problems associated with allocating buffers for these messages and develop a scheme which preserves delivery time guarantees. Issues relating to the implementation of real-time channels are deferred to Chapter 6.

In Chapter 4, we address the problem of selecting routes for inter-process communication in a network with virtual cut-through capability, while balancing the network load and minimizing the number of times that a message gets buffered. The approach taken here is to formulate the route selection problem as a minimization problem, with a link cost function that depends upon the traffic through the link. The form of this cost function is derived based on the probability of establishing a virtual cut-through route. It is shown that this route selection problem is $\mathcal{NP}$-Hard in the general case, so an approximate algorithm

is developed which tries to incrementally reduce the cost by re-routing traffic. The performance of this algorithm is evaluated for two regular network topologies: the hypercube and the C-wrapped hexagonal mesh — example networks for which virtual cut-through switching support has been developed. This chapter also presents an algorithm which can be used to select routes for real-time channels.

Chapter 5 is focused on the problem of broadcasting in mesh networks. A simple extension of the virtual cut-through switching method is proposed, which provides good support for broadcasting in mesh-connected multicomputers. An implementation of this extension (termed a broadcast primitive) for HARTS is also presented. Based on this primitive, a set of broadcast algorithms is developed for the hexagonal mesh topology. These algorithms deliver multiple copies of a message from a source node to every other node in the hexagonal mesh through disjoint paths. They can be used for broadcasting in the presence of faulty nodes/links, even when the identity of the faulty components is not known. It is noted that the 5-BCAST algorithm described in this chapter is an important component of the clock synchronization scheme presented in [RKS90]. The performance of these algorithms is analyzed and compared with the performance of other possible broadcast algorithms.

In Chapter 6 we describe the design and implementation of a communication subsystem for HARTS, which incorporates the ideas presented in the earlier chapters. We first present the kernel for the HARTS NP on which the subsystem is built, and describe how it interfaces with the Application Processors. We then describe the components of the subsystem, which provide different services, like the clock synchronization service. We place special emphasis on the components relating to the implementation of real-time channels and these are described in detail. We also present preliminary results on the performance of the subsystem.

The dissertation concludes with Chapter 7, which reviews the contributions of this dissertation and presents a discussion of future directions for the work presented herein.

CHAPTER 2

PRELIMINARIES

In this chapter, we present the framework for the dissertation, and explain our model for real-time communication. The approach that we take is to design a subsystem which provides various communication related services to applications. In the first section, we present a high-level description of this communication subsystem, leaving the details of the implementation for Chapter 6. Following this, we present the model for communication with timing constraints.

2.1 The Communication Subsystem

The important objectives of a real-time communication subsystem are to deliver messages before certain deadlines and to provide mechanisms for reconfiguration in case of failures of network components. It also offers services such as maintenance of a global time-base using a clock synchronization algorithm, and support for group communication. The global time-base has benefits not only for application tasks, but also several system functions like message scheduling and network management. Also, the subsystem incorporates several functions, like route selection and broadcasting, which are necessary to support these services.

In HARTS, communication functions are the responsibility of the Network Processor (NP). The communication subsystem is designed to be resident on the NP and it is interfaced to the real-time kernel which runs on the Application Processors (APs). The APs currently run the pSOS™ uniprocessor real–time operating system kernel [Sof86]. pSOS serves as the executive on each AP and provides facilities like process management, memory management, event handling and local inter-process communication. In prior work [KKS89], we had extended the pSOS primitives to work in a multiprocessor and multicomputer envi-

™pSOS is a trademark of Software Components Group, Inc.

Interface to the AP		
Services	Network Management	Name Service
	Real-time Channel	Group Channel
	Clock Synchronization	
Transport Level	Request–Reply	User Datagram
Network Level	Datagram Service	
Link Level	Broadcast Service	
	Message Scheduling	

Table 2.1: Communication subsystem structure.

ronment. This system provides pSOS-style communication between nodes and serves as a test-bed for distributed workloads [KS90b]. However, the pSOS communication primitives do not have any notion of timing constraints associated with them. This was one of the motivating forces for the development of a new communication system.

Table 2.1 presents an overview of the services that are provided by this subsystem. This table shows the functions provided at the different layers in the OSI reference model. As is the case in many LAN architectures, some of the higher layers of the reference model are not required. For example, since we are targeting a homogeneous system, presentation level functions are not required. At the link level, we have to manage multiple point–to–point links and take care of packets that are transiting through the node. We also have to prioritize between different packet types and regulate the access to the links. The broadcasting mechanism is also supported at this level. It will be seen that this mechanism can benefit greatly from hardware assistance at the physical level. At the network level, we provide a simple datagram service which can be used by the higher level services and by applications. At the transport level, we support both request–reply and datagram functions. The request–reply function is used by the network management and the name service functions, and it can also be accessed by other functions. The Real-time Channel service is our method for providing time-constrained communication, and it is the topic of the next section. The network management functions include the establishment of real-time channels and handling of link/node failures. The services of this subsystem are made available to processes running on the APs by the interface layer.

As stated above, one of the design objectives for the subsystem is reconfiguration in the case of network failures. Before elaborating on the communication scheme, we would

like to clarify some issues regarding fault-tolerance, including the assumptions that we make.

The Fault Model: Since the system under consideration is not designed for a specific application, it is not clear what fault model would be appropriate for the entire system. Consequently, our approach is to investigate the types of faults that can be handled, at a reasonable cost, at each level or for each operation. We would then analyze the class of faults that the system can tolerate after examining all the levels in detail.

For instance, at the link level, we use a checksumming mechanism as protection against corruption of data in messages. For the NP, the model adopted is that of non-malicious faults where components fail gracefully, that is, failed components do not collude to disrupt the operation of non-faulty components. Thus, we assume that a failure in the NP will manifest itself only as a loss of messages and it can be detected by the other neighboring nodes in the system. Another way of looking at this, from the network viewpoint, is that the faults considered are *timing* or *omission* faults [CASD85]. In [Sch84], Schneider describes a method for building such non-malicious processors.

At the application level, the group communication facility is provided to support fault-tolerance by masking faulty outputs using replication and voting. We do not attempt to recover from process faults; they are just masked by the voting process. For real-time systems, this scheme is considered to be more suitable than a rollback recovery scheme because of the timing constraints involved. The group communication facility is being developed as an extension of the real-time communication scheme presented in this chapter.

Fault Detection and Recovery: In this dissertation, we restrict our attention to detection of, and recovery from, network faults, which can be classified as *link* and *node* failures. One possible approach for fault detection is to periodically exchange status messages. When a failure is detected, it is necessary to reroute the traffic that originally used the failed components. This topic will be covered in greater detail in Chapter 6.

2.2 Real-time Communication

The objective of our work here is to provide and support an abstraction which allows expression of the communication requirements of real-time applications. Specifically, we consider the issue of guaranteeing the delivery of messages with time constraints. The problem of time-constrained communication can be defined as follows.

Definition 2 Time-Constrained Communication: *A message generated at the source station by an application program must be received at the destination station within a given amount of time after its generation at the sending station. Messages which cannot be delivered to their destinations before the given time constraint (deadline) are considered lost.*

The problem of time-constrained communication has been studied by several researchers, since it plays an important role in video- and voice- data transmission over a data network. Recently, it has also been studied in the context of communication in embedded or real-time systems. These efforts have been directed mainly towards designing medium access protocols for multiple-access networks which consider time constraints on messages. The medium access protocols try to implement a distributed scheduling discipline, and the focus has been on developing algorithms which try to minimize message loss. For example, in the case of CSMA/CD networks, the proposed algorithms include the virtual-time based methods [ZR87] and the window based access methods [KSY88, ZSR88]. The survey paper by Kurose *et al.* [KSY84] discusses many of the other proposed techniques. Most of these schemes can be classified as *best-effort* schemes, where the system tries to ensure that most messages meet their deadlines, but it cannot give any guarantees about the delivery times. The performance of these algorithms is demonstrated mainly by simulation using certain stochastic arrival and deadline patterns.

On the other hand, when the system has some information about the arrival pattern of messages, it can try to give guarantees about their delivery times. For example, Strosnider and Marchok [SM89] use a variation of the rate-monotonic scheduling algorithm to control access to a token-ring network. They assign priorities at design time to message sources based on the periodicity of message generation using which they can check for the possibility of deadline overrun. In their work, the time-constraint on periodic messages is implicitly assumed to be the beginning of the next period. Recently, Arvind and others [ARS91a, ARS91b] have proposed a scheme for guaranteed delivery of messages on a multiple access network using a window-based medium access control algorithm. However, their guarantee computation is based on a *local* worst-case analysis where it is assumed that

all other nodes in the network have packets to send. They do not make use of information about actual message generation at other nodes.

The network under consideration here is, however, not a multiple-access network, but has a point–to–point interconnection structure which, as mentioned earlier, has potential for higher performance and reliability than bus/ring structures. In this case, the problem is more complicated than multiple–access networks, because we have to consider delivery time constraints across multiple stages in the network. In this type of network, there is only one source node for any network link, so the issue to be addressed is not one of access to the medium but that of message scheduling in the network nodes. Although it is possible that messages which have to traverse multiple links to reach the destination may suffer from the problem of higher latency, the latency can be made more *predictable* using message scheduling and network flow control. The higher latency need not be a problem for periodic messages, where it is generally assumed that the deadline for a periodic message is the beginning of the next period, which is typically large. There has not been much work reported on the problem of providing time-constrained communication in a local-area point–to–point interconnection network. However, the work by Ferrari, Anderson and others in the DASH and Tenet projects [Fer89, FV90, And88, AF88] dealing with continuous–media communication in wide-area networks is closely related to the work reported here. (See Section 3.5 for differences between this work and ours.)

2.2.1 Message Types

Before elaborating on the topic of time-constrained communication, we will first characterize the types of messages encountered. In a real-time system, there are several classes of messages with different requirements. Guarantees are required for a small but significant number of messages in the system. In the manner of Strosnider [Str88], we classify message traffic into three classes:

1. Alert messages
2. Real-time messages
3. Non real-time messages.

Alert messages are aperiodic and they have strict delivery time constraints. They are considered to represent catastrophic conditions, hence they are allowed to interfere with normal system operations. We perceive that messages in this class would require the transmission of multiple copies on node-disjoint paths to guard against transmission errors.

Real-time messages, which have time-constraints on their delivery time, form the second class. These include both periodic and aperiodic messages, and in either case the time-constraint would be explicitly specified. For periodic messages, the periodicity can be precisely specified and the inter-arrival time will be approximately constant. An important subclass of this are the *Clock* messages which would be exchanged between nodes as part of the clock synchronization algorithm. The arrival pattern for aperiodic messages can be specified using a pessimistic estimate for the minimum inter-arrival time. It is assumed that an occasional loss of messages, due to transmission errors or because of a time-constraint violation caused by the presence of Alert messages, is permissible for these messages. The multiple-path approach can be used to further minimize this type of message loss, if the requirements of the application demand it.

Messages of the third class do not have hard time-constraints, so their scheduling and routing is much more flexible. The system is only required to provide "best-effort" delivery for these messages. These messages can be considered to have advisory time-constraints. That is, the system can discard messages whose time-constraint has been violated. This category also includes messages without any expressed timing constraint, like those found in general-purpose distributed systems.

The communication system has to support all three types of messages. Of these three types, real-time messages pose many problems because they require delivery time guarantees and, unlike Alert messages, they are not infrequent. Hence, we will first look at real-time messages, and then at the other types of messages.

2.2.2 Real-time Messages

Communication between user-level entities in this system can either be connection-oriented or connectionless. In the connection-oriented case, it can be either message stream or byte stream oriented. In the context of time-constrained communication, the communication model has to preserve message boundaries, and so a byte stream model is not suitable. Connection-oriented service is considered more suitable for applications which require guaranteed delivery time for communication [FV90]. The rationale for this choice is that, given an isolated message with an arbitrary deadline, it is very difficult to guarantee its delivery *a priori*. In order to make a guarantee about delivery time for a message, the system requires information about other message sources that can contend for resources with this message. It is therefore necessary to provide a mechanism for describing the characteristics of communication, so that resources can be reserved for real-time connections.

In a connection-oriented service, the connection establishment procedure gives the service provider, in this case the distributed operating system, the opportunity to reserve resources for the connection, and for the user to specify its requirements. Therefore, the abstraction that we use for guaranteed time-constrained communication is one of connection-oriented sequenced messages, which we call a *real-time channel* or simply *channel.* In a bidirectional connection between a pair of user entities, the message generation characteristics may differ substantially for the two directions. Hence, it is preferable to restrict the real-time channel to unidirectional communication, and a bidirectional connection can be composed from a pair of channels.

The resources required by a channel include network bandwidth, buffer space, and message processing bandwidth. The operating system can make resource reservations based on the communication requirements of the user, as specified in the request for the connection. These requirements consist of the source and destination end-points, a description of the message generation process, and the desired end–to–end guarantee for the delivery time. Without a description of the message generation process, the service provider cannot compute the resource requirements and hence it cannot provide a guarantee.

Message Generation Model: We note that a significant portion of the traffic which requires guarantees will be periodic, so our model for message generation is slanted towards periodic traffic. The message generation process is specified in terms of a linear bounded arrival process, a model which was first proposed by Cruz [Cru87]. This model has also been adopted by Anderson and others [ATW+90, AHS90] for continuous media applications, and some of the terminology given below is from [ATW+90]. The arrival process has the following parameters:

maximum message size	S_{max}	(bytes)
maximum message rate	R_{max}	(messages/second)
maximum burst size	B_{max}	(messages)

The model includes the restriction that, in any time interval of length t, the number of messages generated may not exceed $B_{max} + t \cdot R_{max}$, and that the length of each message may not exceed S_{max}. R_{max} is a bound on the message generation rate and its reciprocal, I_{min}, is the minimum (logical) inter-arrival time between messages. The burst parameter B_{max} puts a bound on the allowed short-term variation in the message generation rate, and partially determines the buffer space requirement of the channel. Message generation

which is not periodic can be represented in this model using an estimate of the worst-case inter-arrival time and the average rate of generation.

2.2.3 Channel Specification

As described above, a real-time channel is a network level abstraction similar to a virtual circuit, that represents a unidirectional stream of sequenced messages between end-points on the source and destination nodes. A channel has many attributes that would be used to describe the requirements of the processes which use it. These attributes must express the requirements of each of the message types given in the previous section. Some of the attributes represent parameters required by the lower levels that support the channel abstraction for providing the service, like the message generation process. A description of the channel attributes is given below.

Type	the message class
Source	the source port of the channel
Destination	the destination port of the channel
Length	the maximum message length
Rate	the maximum message rate
Burst	the maximum burst size
Laxity	the bound on the message delivery time
Reliability	the number of copies to be sent, required to guard against transmission errors and message loss

It can be seen that these attributes are sufficient to describe the requirements of the three message types. However, it is not necessary to specify all the parameters. Alert messages are aperiodic and infrequent, so in this case it is not necessary to specify the rate and burst attributes. The laxity and reliability attributes will be specified to reflect the desired time bound and the number of copies. The reliability attribute can also be used for real-time messages when it is necessary to protect against message loss. We will elaborate on this in Chapter 6. For non real-time messages the message generation parameters are not necessary since resources are not reserved for this class.

2.2.4 Channel Establishment

The channel abstraction would be supported as one of the services of the communication subsystem, primarily by the network manager and the message scheduler (see

Table 2.1). The actions taken by the communication subsystem to establish a channel depend upon the type of channel.

Alert messages: These messages are given the highest priority at each service point, even at the risk of violating the deadlines of some real-time messages. This ensures that the delay experienced by these messages is small. Also, since they are infrequent, we can assume that there is no contention for service amongst Alert messages.

When an Alert message channel is created, the source node creates several paths to the destination, as specified in the reliability field. These paths have to be link and node disjoint, so that the message will be delivered even in the presence of congestion and failures. We also choose the paths such that their interaction with the paths of other Alert messages is minimized. Ramanathan and Shin [RS91] show how the multiple copy approach can improve the probability of timely delivery for messages.

Real-time messages: The channel establishment operation is quite complicated in this case, and it is the topic of the next chapter.

Non real-time messages: In this case, channel establishment is a local operation. The parameters of the channel are stored in a session structure, so that they can be used during the actual message transmission. The priority mechanism used for the transmission of these messages is described in Section 3.2.3.

We feel that the channel abstraction is adequate for many tasks, but it may be necessary for some tasks to bypass this interface and get directly to the lower level facilities like the datagram service and the reliable broadcast service. This would be of use to tasks like the time maintenance service. Also, in this way, a request-reply type communication mechanism could co-exist with the unidirectional channel service. A description of these other services can be found in Chapter 6.

CHAPTER 3

REAL-TIME CHANNELS

In this chapter, we will be concentrating on issues related to the design of channels for real-time messages, using which the system provides *a priori* guarantees for message delivery. We will also briefly discuss how best-effort delivery can easily be accommodated into the design. Section 3.1 describes the guarantees provided for messages transmitted on real-time channels. In Section 3.2 we present a solution scheme which is general and works for messages of arbitrary length. Section 3.3 describes a buffer management scheme which is consistent with the message handling scheme described in Section 3.2. Section 3.4 presents extensions to this scheme to help it handle long messages more effectively. We discuss other related work in Section 3.5 and draw our conclusions in Section 3.6. A summary of the notation used in this chapter related to real-time channels is presented in Table 3.1.

3.1 Delivery Time Guarantees

We define the semantics of the end-to-end guarantee on delivery time based on the linear bounded arrival process model which was described in the previous chapter. The *logical generation time*, $\ell_c(m_i)$, for a message m_i on channel M_c can be defined as

$$\begin{aligned} \ell_c(m_0) &= t_0^c \\ \ell_c(m_i) &= max\{\ell_c(m_{i-1}) + I_{min}^c,\ t_i^c\} \end{aligned}$$

where t_i^c denotes the actual generation time of m_i. If Δ_c is the end-to-end delay for the channel, the system *guarantees* that any message m_i will be delivered to the destination node by time $\ell_c(m_i) + \Delta_c$. In other words, when the inter-arrival time between messages is at least I_{min}^c, the system guarantees that each message in the channel incurs a delay of at most Δ_c seconds. However, messages which arrive in a burst, where the inter-arrival time is less than I_{min}^c, may suffer a larger delay since the guarantee is given with respect to the logical arrival time. This possible increase in delay is a consequence of regulation: arrivals

S^c_{max}	The maximum message size for channel M_c.
R^c_{max}	The maximum message rate for channel M_c.
I^c_{min}	The minimum inter-arrival time for channel $M_c = 1/R^c_{max}$.
Δ_c	The maximum end-to-end delay for channel M_c.
$d_{c,a}$	The maximum delay for a message on channel M_c at node a.
$\ell_c(m_i)$	The logical generation time for the ith message on channel M_c.
$\ell_{c,a}(m_i)$	The logical arrival time at node a for the ith message on channel M_c.
t^c_i	Actual arrival time of the ith message on channel M_c at some node.
C_i	The maximum service time required for messages on channel M_i.
p_i	same as I^i_{min}.
r_i	Response time for a channel M_i at some link.
C_{pkt}	The service time for a single maximum-length packet.
C_{grp}	The service time for a packet group.
H_k	Horizon for the link k. For example, $H_{a,b}$ denotes the horizon for link (a, b).

Table 3.1: Notation

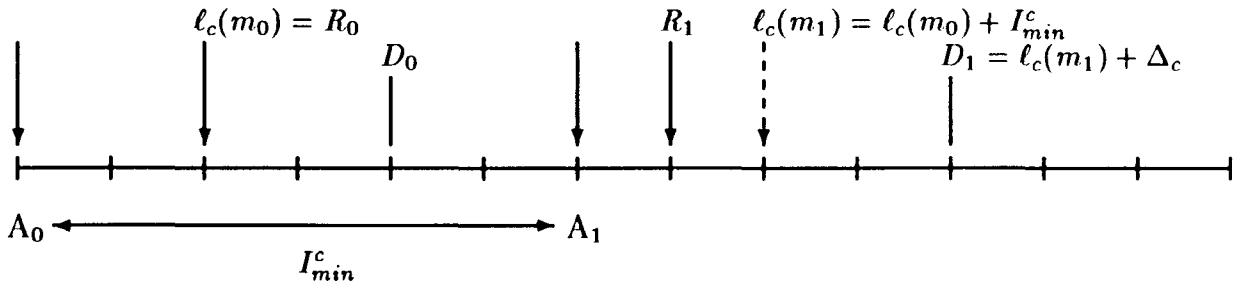

Figure 3.1: Delivery time guarantee.

at each node have to be regulated in order to prevent burst arrivals on one channel from affecting the guaranteed delay of messages belonging to other channels.

This definition of the message delivery guarantee is well suited for periodic message generation. For example, consider a periodic task which generates a message at the end of its execution. The generation time of the message depends upon the *response time* seen by the task, and this can vary from one invocation to another. If these messages are transmitted on a real-time channel, the worst-case delivery time of a message can be linked to the *arrival time* of the periodic task. Figure 3.1 shows two instances of a periodic task which occur at time A_0 and A_1 respectively. The messages form a part of a real-time channel M_c with inter-arrival time I^c_{min} and delay Δ_c. The first instance of the task experiences its worst-case response time R, so the message is released at time R_0. The second instance of the task experiences a response time $(R_1 - A_1) < (R_0 - A_0)$. Hence, the interval between the generation time of the first and second messages is less than I^c_{min}, i.e., the second message arrived early. The logical generation time $\ell_c(m_1)$ assigned to the second message is therefore greater than its actual arrival time. However, it can be seen that the generated messages will always be delivered by time $R+\Delta_c$ from the beginning of the task period. This property can be used in any off-line schedulability analysis for the system.

This chapter deals with problems connected with computing a guaranteed end-to-end delay for messages belonging to a channel, and the scheduling of messages to achieve this goal. These two problems are related because it is necessary to understand the scheduling environment in order to compute a guarantee. In the next section, we will first present a simple channel establishment procedure in which the problem of computing an end-to-end guaranteed delay is reduced to the problem of guaranteeing the worst-case delay for a single station/node.

1. Select a source–destination route for the channel.
2. Compute the worst-case delay for a message on each link on the route. In this computation, it is necessary to ensure that the new channel does not affect the guaranteed delivery times of existing channels. Also compute the buffer requirement for this channel.
3. Compute the sum of the worst-case delays which were determined in Step 2, and check whether it is less than the user-specified delay. This is the channel establishment test.
4. If the channel can be established, divide the user-specified delay among the links on the route based on their worst-case delay for the message. Adjust the buffer requirements based on these allocated delays.

Figure 3.2: Channel Establishment Procedure

3.2 Basic Solution Approach

There are two distinct phases in handling real-time channels: channel establishment and run-time message scheduling. The channel establishment phase is outlined in Figure 3.2, and begins with the selection of a route for the channel. There are two alternatives for routing packets in the network: dynamic routing and static routing. Dynamic routing offers the advantage that it can adapt to the network load and can reduce the average delivery delay for messages. However, it is very difficult to make any guarantees on message delivery for a channel based on dynamic routing, in which a message can use one of several alternate routes through the network. Therefore, we use static routing for messages belonging to real-time channels. We will assume that a route has been selected for the channel, possibly using a scheme like the one presented in [KS90a], and concentrate on the computation of guarantees and on buffer management.

The worst-case delay for a message at a link depends upon the scheduling algorithm used and the other channels which use the link. The link may also be used by messages without time constraints, but the effect of these messages on the worst-case delay is limited (i.e., restricted to a single packet delay if packet transmission cannot be aborted while in progress) because they would belong to a lower priority class. Hence, we can restrict our attention to messages belonging to real-time channels while calculating the worst-case delay. The scheduling environment that we encounter in message scheduling is one of independent, possibly periodic, message arrivals which have deadlines associated with them. The message deadline may be related to its period, but it is not necessarily the beginning of the next period. There are several approaches to scheduling these messages, which can be categorized

as fixed priority or dynamic priority algorithms. For example, Earliest Due Date [Der74] is a dynamic priority algorithm, whereas rate monotonic scheduling [LL73] is a fixed priority algorithm. We will now discuss the problems associated with the use of these algorithms for message scheduling and channel establishment.

Deadline Scheduling: The Earliest Due Date (EDD) algorithm schedules messages in the order of their deadlines, with higher priority given to messages which have closer deadlines. Liu and Layland [LL73] have shown that EDD is optimal for preemptive scheduling of periodic tasks when the task deadline is equal to the beginning of its next period, and that a feasible schedule exists whenever the total utilization is less than one. However, there is no similar result (based on utilization alone) available when the task deadlines are not related to their periods. The main drawback of EDD scheduling is that computation of guarantees is difficult, since the priority of a task depends upon the relative order of arrival of the tasks. A multi-class version of the EDD algorithm has been used for scheduling real-time messages by Ferrari and Verma [FV90, Fer89], who also present sufficient conditions for the existence of feasible schedules. Their approach will be discussed in greater detail in Section 3.5.

Fixed Priority Scheduling: Scheduling decisions can be based on a fixed (static) priority scheduling algorithm where messages are processed and transmitted in the order of priority. For example, in rate-monotonic scheduling the priority assigned to a channel is related to the frequency of occurrence of messages on that channel. For any priority assignment scheme, if message arrivals on all the channels are assumed to be strictly periodic, we can determine whether the set of channels is schedulable. This is done by computing the worst-case response time for each message using a *critical time zone* analysis similar to the one used by Liu and Layland [LL73], and verifying that the worst-case response time is less than the delay assigned to that channel. The details of this scheme for analysis can be found in Appendix 3.A.

There are some problems with priority based scheduling and the computation of guarantees when we consider multiple stage service, like service for a message which has to traverse multiple links. The response time for the message varies depending upon the arrival time of other messages at a node. Therefore, even if messages are generated with a fixed inter-arrival time at the source, the inter-arrival time for the message at the next service point is not constant. Early arrivals are also possible due to burstiness in the message generation at the source. A message which arrives early at a node can cause a lower priority message to miss its deadline. Figure 3.3 shows the results of preemptive priority scheduling

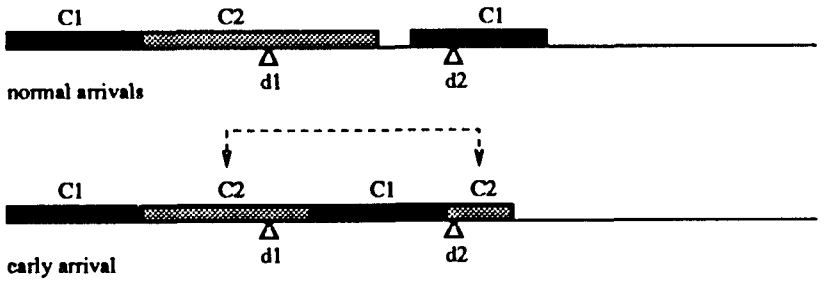

Figure 3.3: Effect of early arrivals.

for two message streams, M_1 and M_2, with service requirements C_1 and C_2, where M_1 has higher priority than M_2. The figure shows how the early arrival of a message on M_1 can cause a message on M_2 to miss its deadline d_2. Note that a similar effect can be observed even when a non-preemptive discipline is used. One solution for this problem is to "hold back" the early arrival and not consider it for transmission until its scheduled arrival time. However, this scheme involves the setting and resetting of timers and is expensive to implement. Also, it implies that the message cannot make the best progress possible on lightly loaded links.

3.2.1 Proposed Algorithm

The discussion on the problems of deadline and fixed priority scheduling in a multi-hop communication system suggests the use of a combination of deadline and fixed priority scheduling. Our channel establishment scheme uses fixed priority scheduling for computing the delay, but we use a form of EDD as the run-time message scheduling algorithm. When a channel is to be established, for each link on its route, we estimate the worst-case response time for a message based on fixed-priority scheduling using the method outlined in Appendix 3.A. The response time analysis is based on an approximation of preemptive scheduling. Pure preemptive scheduling cannot be used in the context of message scheduling, because, if the transmission of a message is interrupted, the message is lost and has to be retransmitted. To achieve the benefits of preemptive scheduling, the message has to be split into packets so that message transmission can be interrupted at the end of a packet transmission, without loss. (This is analogous to allowing an interrupt at the end of an instruction execution.) Therefore we consider a message to be a set of one or more packets, where the packet size is bounded. Packet transmission is non-preemptive, but *message* transmission can be considered to be preemptive.

The priority assigned to the new channel at a link depends upon the characteristics of the other channels going through the link. The total response time, which is the sum of the response times over all the links on the route of the channel, is checked against the maximum permissible message delay and the channel can be established only if the latter is greater. In this case, the permissible message delay is split proportionally among the different links. While using this procedure, it is necessary to ensure that the new channel does not affect the guaranteed delivery times of existing channels. This is taken care of in the priority assignment algorithm.

3.2.2 Priority Assignment

The worst-case response time computation for a new channel at a link on the route requires an assignment of priorities for existing channels that use this link. This priority assignment problem can be defined formally as follows. Let $\{M_i = (C_i, p_i, d_i), i = 1, \ldots, k\}$ be the set of k existing channels through a link ℓ, where C_i is the maximum service time required for messages of channel M_i on this link, $p_i = I^i_{min}$ is the message inter-arrival time, and d_i is the permissible delay which has been assigned to the channel for this link. The service time requirement C_i is proportional to the product, $S^i_{max}.R^i_{max}$, of the maximum message size and the maximum message rate for the channel M_i. Given a new channel $M_{k+1} = (C_{k+1}, p_{k+1})$ to be established (the delay bound need not be specified on a per-link basis), we have to find a priority assignment for this augmented set of channels such that the response time r'_i computed for a channel under this priority assignment satisfies the constraint $r'_i \leq d_i, i = 1, \ldots, k$. An assignment which satisfies this constraint is called a *feasible priority assignment.* Moreover, we would like to find a feasible priority assignment which will minimize the response time r'_{k+1} for the new channel M_{k+1}. By finding the minimum feasible response time on each link, we can improve the chances of satisfying the channel establishment test.

Consider the procedure D_Order, shown in Figure 3.4, which assigns priorities and computes the response time for all channels, including the one to be established. This procedure works under the condition that, for all channels, $d_i \leq p_i$. That is, the worst-case delay at each link for any channel does not exceed its inter-arrival time. Note that the total end-to-end delay can exceed the inter-arrival time of the channel.

We can prove that, when $d_i \leq p_i, \forall i$, assignment procedure D_Order is optimal in the sense that the computed response time r'_{k+1} is the minimum possible for any feasible priority assignment. Lemma 1 proves that the priority assignment based on delays is optimal

1. Arrange the channels in ascending order of their associated delay d_i.
2. Assign the highest priority to the new channel M_{k+1}. Assign priorities to the other channels based on this order, with high priority assigned to channels with small delays.
3. Compute the new (worst-case) response times r_i' for the existing channels based on this priority assignment.
4. In the priority order, find the smallest position q such that $r_i' \leq d_i$ for all channels with position greater than q.
5. Assign priority q to the new channel and compute the response time r_{k+1}'.

Figure 3.4: Assignment Procedure D_Order

in the sense that if there is any feasible priority assignment for the channel set, there is a feasible priority assignment based on delays. This is in fact a generalization of the optimality result for rate-monotonic scheduling [LL73]. In their model, Liu and Layland assumed that the deadline for an instance of a periodic task is the release time of the next instance, that is, $d_i = p_i$. The rate-monotonic algorithm assigns priorities to periodic tasks based on their periodicity, with high priority assigned to tasks with high periodicity. The optimality result for this scheduling algorithm can be derived from Lemma 1 by substituting $d_i = p_i, \forall i$. It has been brought to our attention that a result similar to Lemma 1 has also been proved by Leung and Whitehead [LW82].

Lemma 1 *Consider the set of channels* $\{M_i = (C_i, p_i, d_i), i = 1, \ldots, k\}$ *through a link. Assume that there exists a feasible fixed priority assignment for the channels such that the computed response time for each channel satisfies the constraint* $r_i \leq d_i \leq p_i, \forall i$. *Then, the priority assignment* P^D, *based on an increasing order of delays with high priority assigned to channels with small delays, is also a feasible priority assignment.*

Proof: We can assume, without loss of generality, that the set of channels is ordered by priority in accordance with a feasible priority assignment P. That is, $i < j$ implies that M_i has higher priority than M_j under P. Now, if $d_i \leq d_j, \forall i < j$, our proof is complete. Otherwise, there exists at least one pair of adjacent channels, M_i and M_{i+1} such that $d_i > d_{i+1}$. In this case, we want to prove that priorities of these two channels can be swapped to yield another feasible assignment, P'.

Let r_i and r_{i+1} be the worst-case response times of the two channels M_i and M_{i+1}, computed under the priority assignment P using the scheme shown in Appendix 3.A.

Then, since $d_{i+1} < d_i \leq p_i$ and $r_{i+1} \leq d_{i+1}$, $r_{i+1} < p_i$, and so r_{i+1} includes the service time for exactly one instance of M_i. Under P', the response time for M_{i+1} is reduced, so the constraint $r'_{i+1} \leq d_{i+1}$ is trivially satisfied. Also, by the definition of the worst-case response time (see Eq. (3.8) in Appendix 3.A), r_{i+1} contains exactly one instance of M_{i+1}. The response time r'_i for M_i under P' also has to contain one instance of M_i and M_{i+1}, and it can be seen that $r'_i = r_{i+1} \leq d_i$.[1] For all other channels we have $r'_j = r_j \leq d_j$, and hence P' is a feasible priority assignment.

It is evident that by a sequence of pairwise exchanges, we can reach P^D and show that it is a feasible assignment. ∎

Theorem 1 *Assignment procedure D_Order yields the smallest response time for the new channel, M_{k+1}.*

Proof: Suppose there is some other feasible priority assignment P which yields a smaller response time. If r_{k+1} is the response time for M_{k+1} under P, we can set $d_{k+1} = r_{k+1}$. Consider any channel M_j which has lower priority than M_{k+1}. Then, $r_{k+1} < r_j \leq d_j$ and hence $d_{k+1} < d_j$. Now, by Lemma 1, the priority assignment P^D based on the channel delays is also a feasible assignment. Under this assignment, $d_{k+1} < d_\ell$ implies that M_{k+1} has priority over M_ℓ. Therefore, any channel which has priority lower than M_{k+1} under P also has priority lower than M_{k+1} under P^D. Also, a channel which has priority higher than M_{k+1} under P could have priority lower than M_{k+1} under P^D. This means that the response time for M_{k+1} under P^D will be no greater than the response time under P. Contradiction. ∎

D_Order can be implemented efficiently by storing some additional information about the existing channels. We can maintain the channels in ascending order of permissible delays, so the cost of sorting can be ameliorated. The computation of the response time for all channels is then the most important operation. We can eliminate duplicate computations by computing the response times in ascending order. For example, $r_{i+1} \geq r_i + C_{i+1}$, and we can use this as the starting point in the scan. Also, in the computation, if $r'_i > d_i$ then the computation for r'_i can be aborted.

In the last stage of the channel establishment procedure, delays will be assigned to the new channel, for each link on the source–destination route. For any link, the assigned delay for that link, d_{as}, has to be such that $d_{as} \geq r'_{k+1}$. It can be shown that if channels are

[1] Note that this result holds only because $r_{i+1} \leq p_{i+1}$, so that when we switch priorities only one instance of M_{i+1} is included in the response time for M_i.

allotted priorities based on this assigned delay, the resulting priority order is still feasible. The new priority order for the channels is not necessarily the one used to compute the response time r'_{k+1} because there may exist a channel with $r'_{k+1} \leq d_i < d_{as}$.

3.2.3 Run-time Scheduling

A node in the system can have several incoming and outgoing links connected to it. These links can operate in parallel, so each outgoing link is considered as a separate entity for scheduling. Messages are composed of packets, and packets carry information about the message, and the channel, to which they belong. When a packet arrives at a node, or is generated in the node, it is dispatched to the appropriate outgoing link. Suppose the arriving packet belongs to the ith message on the channel M_c. All packets belonging to the ith message would be assigned the same *logical arrival time*, $\ell_c(m_i)$, which is the logical arrival time for the message. For the source node s of channel M_c, $\ell_{c,s}(m_i)$ is defined exactly in the same way as the logical generation time in Section 3.1, where t_i^c denotes the generation time of message m_i.

$$\begin{aligned} \ell_{c,s}(m_0) &= t_0^c \\ \ell_{c,s}(m_i) &= max\{(\ell_{c,s}(m_{i-1}) + I_{min}^c),\ t_i^c\}. \end{aligned} \tag{3.1}$$

For nodes other than the source node, the logical arrival time at the node is based on the logical arrival time of the message at the upstream node. The information about the logical arrival time is carried in the packet, as a part of the packet header. Consider two adjacent nodes a and b, sharing a link (a, b) which is a part of the route for channel M_c. The logical arrival time for m_i at node b, $\ell_{c,b}(m_i)$, is defined as

$$\ell_{c,b}(m_i) = \ell_{c,a}(m_i) + d_{c,a} \tag{3.2}$$

where $d_{c,a}$ is the worst-case delay for messages on channel M_c at node a. It can be seen that the logical arrival time of a message at any node is ultimately based on its logical arrival time at the source node. This definition of the logical arrival time is feasible because the nodes in the system have synchronized clocks and the maximum skew between clocks on different nodes is small compared to the message delivery delays [RKS90].

Alternatively, it is conceivable to define $\ell_{c,b}(m_i)$ based only on the *observed* arrival time of messages at node b, without carrying any timestamps in the packet headers. However, this type of scheme has problems when we consider multiple-packet messages. For example, $\ell_{c,b}(m_i)$ can be defined based on the arrival time of the *first* packet of m_i. But

then, it is possible that the last packet of m_i may arrive up to (approximately) $d_{c,a}$ time units later than the first packet, if we consider the worst-case delay for m_i over just one link (a, b). When we consider delays for m_i over multiple links, the separation between the first and last packets can increase further. In this case, the last packet can be assigned a logical arrival time which is *beyond* its local deadline. This can result in some problems for the deadline scheduler. On the other hand, if $\ell_{c,b}(m_i)$ is defined based on the arrival time of the *last* packet of m_i, the message cannot be processed until the last packet has arrived. Moreover, it would be necessary to guard against loss of the last packet of a message, possibly using message timers.

The logical arrival time that is assigned to messages is used by the message scheduler, which uses a variation of the multi-class EDD algorithm. In the following description, the second subscript has been dropped from the notation for the logical arrival time, since we are dealing with a single node. The scheduler maintains three queues for each outgoing link, corresponding to three service classes.

Queue 1 Packets belonging to real-time channels with $\ell_c(m_i) \leq$ current_time, arranged in the order of increasing deadlines.

Queue 2 Other packets arranged in an appropriate priority order.

Queue 3 Packets belonging to real-time channels with $\ell_c(m_i) >$ current_time, arranged in the order of increasing logical arrival time.

Queue 1 and Queue 3 contain packets which belong to real-time channels, while Queue 2 contains all other types of packets. Queue 1 contains *current* real-time packets, which have to be scheduled in the order of their deadline, hence it is organized by increasing packet deadlines. Current real-time packets are those packets whose logical arrival time is less than the current clock time at the node. Queue 2 contains packets for which no guarantees are given, and the ordering depends upon the message type. For example, packets belonging to messages which request "best-effort" type service would fall in this category, and they would be arranged in the order of increasing deadlines. The service provided for this class of packets is improved by giving them priority over real-time packets which are not *current*. Since they are not pertinent to the current discussion, we will not elaborate on the treatment of this class any further. Packets in Queue 3 are those which have arrived early, either because of burstiness in the message generation or because they encountered delays which were smaller than the budgeted worst-case delays at some upstream nodes. These packets are stored in

1. find $\ell_c(m_i)$, the logical arrival time of the message to which this packet belongs.
2. set ℓt(packet) = $\ell_c(m_i)$.
3. set the deadline for this packet to $\ell_c(m_i) + d_c$.
4. **if** (ℓt(packet) $\leq$ current_time)
   insert packet into Queue 1
   else
   insert packet into Queue 3
5. invoke the dispatcher.

Figure 3.5: Processing on Packet Arrival

the order of their logical arrival time because they have to be transferred to Queue 1 as they become current.

The actions taken when a real-time packet arrives are shown in Figure 3.5, whereas non real-time packets are simply inserted into Queue 2. The logical arrival time is determined based on the channel to which the packet belongs and the sequence number of the message of which it is a part. This is the logical time, $\ell t()$, of a packet. The deadline for the packet is set to $\ell_c(m_i) + d_c$, where d_c is the worst-case delay guaranteed for channel M_c at this node. Since all packets in a message have the same $\ell_c(m_i)$ and the same d_c, they will all have the same deadline. The packet is then inserted into Queue 1 or Queue 3, depending upon whether it is current or early. Lastly, the dispatcher is invoked.

The dispatcher, which is shown in Figure 3.6, first checks whether any real-time packets in Queue 3 have become current and transfers such packets to Queue 1. If the link is idle, it examines the queues in the order of priority looking for a packet to transmit. Packets in Queue 3 are considered for transmission only if their logical time is within the *horizon*, H_k, for the link. The horizon is link-dependent and is used for flow control, as explained in Section 3.3. These packets are scheduled in the order of logical time, primarily because they are queued in that order. Also, this ordering makes it easy to identify packets which are within the horizon (and can be considered for transmission). The dispatcher is also invoked upon completion of transmission of a packet on the link. In some situations, it is possible that Queues 1 and 2 are empty and Queue 3 only contains packets which are ineligible for transmission. In such cases, a timer can be used to trigger the dispatcher at the appropriate future instant.

```
0.  Examine Queue 3. Transfer those packets which have ℓt(packet) < current_time, to Queue 1.
1.  if (link idle)
2.      if (Queue 1 nonempty)
3.          start transmission of head(Queue 1)
4.      else if (Queue 2 nonempty)
5.          start transmission of head(Queue 2)
6.      else if (Queue 3 nonempty)
7.          if (ℓt(head(Queue 3)) < current_time + H_k)
8.              start transmission of head(Queue 3)
9.          end
10.     end
11. end
```

Figure 3.6: Dispatcher

Proof of Correctness

We now have to establish that this run-time scheduling scheme will conform with the guarantees computed using fixed message priorities. This is stated formally in Theorem 2 below.

Theorem 2 *Consider a link containing a set of real-time channels which satisfy the following properties.*

1. The channels are assigned delays for this link using the procedures in Section3.2.1 (Figures 3.2 and 3.4).
2. Arriving messages are assigned logical arrival times using Eqs. (3.1) and (3.2).
3. Messages are scheduled using the procedures in Figures 3.5 and 3.6.

Then, all messages arriving on real-time channels meet their delay requirements on the link.

Proof: We approach this in two stages. In the first stage, we show that if we use this scheduling scheme on a channel set (which is schedulable) that has strict periodic arrivals satisfying the inter-arrival time constraints, then no message will miss its deadline. In the second stage we consider the effects of deviations from the periodic behavior and show how our scheme can accommodate them.

Stage 1: The system times for channels have been computed based on priority scheduling, taking into account the maximum rate of arrival of messages; and these times satisfy the local deadlines for each channel. Hence, when arrivals are all periodic and conform to the inter-arrival time constraints, a feasible schedule based on priorities exists. Dertouzos [Der74] has shown that this implies the existence of a feasible EDD schedule for this message set, which is based on the deadlines assigned to the channels. In this case, the logical arrival time of each message is the same as its arrival time and all arrivals go into Queue 1. Since this queue is scheduled using the EDD algorithm, and a feasible EDD schedule is possible, the delay requirements of all messages will be satisfied.

Stage 2: Consider a message m_j belonging to channel M_c, which arrives at the source node before its logical arrival time, that is, $t_j^c < \ell_c(m_j) = \ell_c(m_{j-1}) + I_{min}^c$. This message would be assigned a deadline, $D_j = \ell_c(m_j) + d_c$, corresponding to its logical arrival time and would be inserted into Queue 3. There are two possibilities to consider: 1) m_j is transmitted directly out of Queue 3, and 2) m_j is subsequently transferred to Queue 1. In the first instance, we can conclude that m_j was transmitted before time $\ell_c(m_j)$ and hence it was transmitted before its deadline. In the second case, some packets of m_j were transferred to Queue 1. Since Queue 3 is scanned at the end of every packet transmission, this transfer will occur by time $\ell_c(m_j) + C_{pkt}$, where C_{pkt} is the transmission time for a single maximum-length packet. By the argument given in Stage 1, we can assert that, if the message m_j arrived into the system at its logical arrival time, a feasible EDD schedule exists by which all messages will meet their deadline. Note that the computed response time for the channels includes C_{pkt}. Hence, the packets of m_j which were transferred to Queue 1 will be transmitted before their deadline. The same argument can be applied even when multiple messages arrive prior to their logical arrival time.

If a message m_i arrives late at the source node, that is, $t_i^c > \ell_c(m_{i-1}) + I_{min}^c$, it will be assigned a logical arrival time $\ell_c(m_i) = t_i^c$ and a deadline, $D_i = \ell_c(m_i) + d_c$. It will then be inserted into Queue 1. Also, future message arrivals on this channel will be assigned logical times based on $\ell_c(m_i)$. This message cannot increase the maximum system time for any other messages, since the system times for these messages were computed by considering the maximum possible arrival rate for messages on channel M_c. Hence, a feasible priority-based schedule exists in this case, and consequently, a feasible EDD schedule also exists. ■

3.3 Flow Control and Buffer Management

Buffer space has to be reserved for real-time channels at the source, destination, and intermediate nodes in order to prevent buffer overruns and consequent loss of messages. During channel establishment, the user has to specify the burstiness in message generation, B_{max}, as part of the channel specification. This burstiness determines the buffer requirement at the source node of the channel. If $d_{c,s}$ is the worst-case delay guaranteed for channel M_c at the source node, the buffer requirement can be expressed as $(B^c_{max} + d_{c,s} \cdot R^c_{max})S^c_{max}$, where the $d_{c,s} \cdot R^c_{max}$ term accounts for new arrivals during the system (response) time of a message on the channel. It is necessary for the source node to provide this buffer space because the client can legitimately produce messages at this rate, and, if buffer space is not provided, messages could be lost due to non-availability of buffers. Also, the source node is responsible for making sure that message generation for M_c adheres to the channel specification. It can refuse to accept messages belonging to M_c when the actual generation rate exceeds the specified bounds.

Intermediate nodes also have to provide buffer space for M_c, but this need not depend upon B^c_{max} if a flow-control mechanism is employed between nodes. The buffer reservation scheme and the flow-control mechanism are intimately related, since flow control can be used to regulate the burstiness of message arrivals at intermediate nodes. The flow-control mechanism considered here operates by holding back some of the messages which arrive before their logical arrival time. If flow control were not used, messages could zip through lightly loaded nodes and back up at heavily loaded nodes, thereby causing buffer space problems at the heavily loaded nodes. An extreme case of flow control is when all messages which arrive early are held back, that is, a message is considered for transmission only when $\ell_c(message) \leq$ current_time. In this extreme case, an intermediate node, k, can rely on flow control to eliminate the burstiness and need only provide buffer space proportional to $d_{c,k} \cdot R^c_{max} \cdot S^c_{max}$. However, this strict regulation can result in unnecessary delays for messages when the network is lightly loaded. By allocating more buffer space, we make it possible for messages to go quickly through lightly loaded nodes.

The logical arrival time assigned to a message depends upon the burstiness in the arrival process. Hence, the bound on burstiness can also be interpreted as a *horizon* for the logical arrival time. Burstiness can be measured by the equation,

$$B = (\ell_c(\text{message}) - \text{current\_time})/I^c_{min}$$

which can be rewritten as

$$\ell_c(\text{message}) = \text{current\_time} + B \cdot I^c_{min}.$$

A bound on the burstiness, $B \leq B_m$, can therefore be interpreted as $\ell_c(\text{message}) \leq \text{current\_time} + B_m \cdot I^c_{min}$. Packets belonging to messages with logical arrival time greater than this horizon may be discarded by the node. In other words, the horizon, $B_m \cdot I^c_{min}$, determines the buffer requirement for a channel at an intermediate node.

A node can compute the horizon for a link, based on the available buffer memory, whenever a new channel is to be established with that link. This horizon is passed back to the upstream node, since that node is responsible for flow control on the link. The upstream node has to make sure that it transmits forward only those messages which have logical time, at that node, within the horizon (packets which belong to a single message all have the same logical time). It is possible to use a different horizon for each channel, but this can create problems for the run-time scheduler because it is then forced to examine all the packets in Queue 3 individually to check their eligibility for transmission. Use of a single horizon means less flexibility in buffer management, but it is more suitable for run-time scheduling. The buffer space required for a channel is computed as follows.

Consider two adjacent nodes a and b, sharing a link (a, b) which has a horizon $H_{a,b}$. Consider a message p arriving at node b on channel M_c, using link (a, b). Since node a uses $H_{a,b}$ for flow control on link (a, b), we can deduce that the logical time for the message at node a must be $\ell_{c,a}(p) \leq \text{current\_time} + H_{a,b}$. The logical time assigned to the message at node b is $\ell_{c,b}(p) = \ell_{c,a}(p) + d_{c,a}$, where $d_{c,a}$ is the assigned worst-case delay for M_c at node a. Node b will not contain any messages belonging to M_c which have $\ell_{c,b}(m) < \text{current\_time} - d_{c,b}$, where $d_{c,b}$ is the worst-case delay assigned to the channel at this node. This is because messages on M_c are guaranteed to leave b within $d_{c,b}$ time units from their logical arrival time. Hence, node b can only contain messages belonging to M_c which have logical time in the range

$$\text{current\_time} - d_{c,b} \leq \ell_{c,b}(p) \leq \text{current\_time} + H_{a,b} + d_{c,a}.$$

The buffer space required to hold these packets is then given by:

$$\lceil (H_{a,b} + d_{c,a} + d_{c,b})/I^c_{min} \rceil \cdot S^c_{max}.$$

From this equation, it is clear that the minimum buffer space required for a channel is $S^c_{max} \cdot \lceil (d_{c,a} + d_{c,b})/I^c_{min} \rceil$, which is when $H_{a,b} = 0$. When the horizon H for a link is very

large, the maximum buffer space required for a channel is bounded by $S^c_{max} \cdot (\lceil \Delta_c / I^c_{min} \rceil + B^c_{max})$, where Δ_c is the end–to–end delay for the channel.

The overall buffer space requirement of a node is the sum of the buffer space required for all channels going through the node. When a new channel is being established, the node first tries to accommodate the new channel by computing the buffer space based on the existing horizon. If the available buffer space is less than the requirement, the node then reduces the horizon for that link. Reducing the horizon for a link has the effect of reducing the buffer space requirement of *all* channels using the link, so the amount of buffer space available increases.

3.4 Extensions for Long Messages

The channel establishment scheme, as presented, can handle messages of any size. However, the response time obtained for a long message can be substantial, and since the latency for message delivery is determined by the cumulative response time over the links in the route, it can grow very large. The underlying problem here is that store–and–forward delays grow with the size of the message. Long messages also pose problems for buffer management at intermediate nodes as these nodes have to reserve sufficient buffer space to hold the entire message. Since the number of channels going through an intermediate node can be large, the buffer space requirement of transit messages is an important issue.

Consider a request for a channel $M_m = (C_m, p_m, \Delta_m)$, where Δ_m is the end–to–end delay for M_m, which has to traverse L_m links. In order to satisfy the end–to–end delay, the sum of the worst-case delays on the L_m links should be no greater than Δ_m. Hence, *on the average,* the worst-case delay at a link should be within Δ_m / L_m. It can be seen that as L_m increases, the delay available at each link reduces and it may not be possible to satisfy the delay requirement. Also, the buffer space required at each link for M_m depends upon C_m (and on the burstiness parameter).

The delay and buffer problems of long messages can be alleviated by splitting these messages into *packet groups* and "pipelining" the transmission of these packet groups[2]. A message on channel M_m can be split into N_m packet groups, each with size $C_{grp} = C_m / N_m$ and delay d_{grp}. The transmission of these packet groups will be in sequence, as shown in Figure 3.7. This figure shows a message split into 2 packet groups, C^1_{grp} and C^2_{grp}, and transmitted over three links. Transmission of each packet group has to complete within an

[2]We use the term packet group to distinguish this entity from a physical packet. A packet group can contain one or more physical packets.

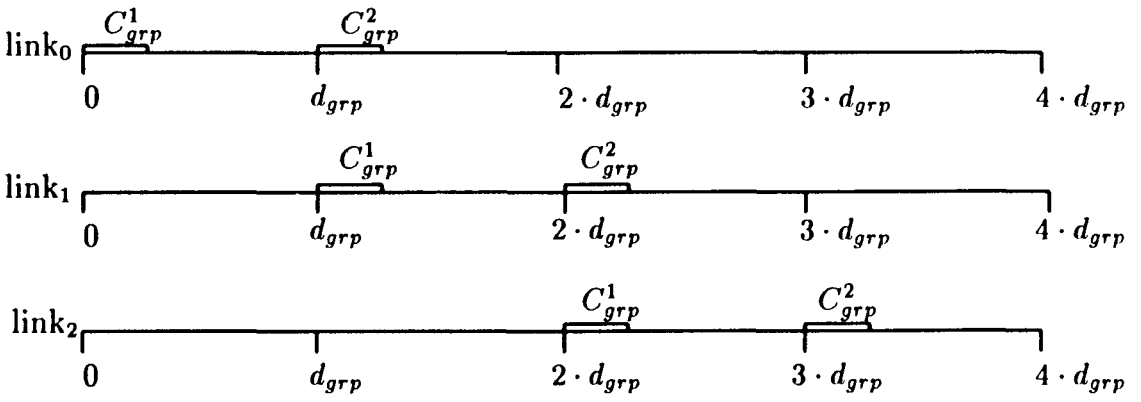

Figure 3.7: Packet group illustration.

interval of length d_{grp}. Here d_{grp} represents the delay in any stage of the pipeline, and it has to exceed (or equal) the maximum response time for the packet group over the links in the route. The pipelining mechanism enables us to bring out the overlap between the transmission of packets belonging to a message of M_m on different links of the route. The end-to-end delay for a message on M_m is then $d_{grp}*(N_m+L_m-1)$ ($= 4d_{grp}$ in the example). Hence, the constraint that d_{grp} has to satisfy is:

$$d_{grp} \leq \Delta_m/(N_m + L_m - 1). \tag{3.3}$$

Selection of N_m: When packet groups are used, a larger value for N_m is preferable from the standpoint of the delay computation. The ratio C_{grp}/d_{grp} defines the fraction of time required to process the packet group to the packet period, and it is desirable to make this small. This ratio, when expressed in terms of the parameters of M_m, is $(C_m/\Delta_m) \cdot (N_m + L_m - 1/N_m)$, a monotone decreasing function of N_m. Also, the buffering problem is reduced because intermediate nodes now have to provide buffer space based on the size of the packet group instead of the message size. To increase flexibility in buffer management, it appears that we should increase the number of packet groups, N_m, as much as possible, preferably to have only one packet per packet group.

In this pipelining scheme, it is necessary that all packet groups of a message should be transmitted from a link before the arrival of the next message on the channel. This imposes a restriction on the possible values for the number of packet groups.

$$d_{grp} \cdot N_m \leq p_m \tag{3.4}$$

Using Eqs. (3.3) and (3.4) we get

$$(\Delta_m/(N_m + L_m - 1)) \cdot N_m \leq p_m \tag{3.5}$$

This equation can also be rewritten as a constraint on N_m, whose solution depends upon the relative values of Δ_m, p_m, and L_m:

$$1/L_m \leq N_m/(N_m + L_m - 1) \leq p_m/\Delta_m. \tag{3.6}$$

There are three cases to be considered:

$\Delta_m < p_m$: In this case, N_m can take any value. However, for large N_m, d_{grp} is small and this implies that the priority assigned to the channel is high. This can interfere with the resource requirements of other channels, and result in the rejection of subsequent channel establishment requests.

$p_m < \Delta_m \leq Lp_m$: The optimal choice for N_m is the largest integer which satisfies Eq. (3.5). For example, if $\Delta_m = 2p_m$ in the example of Figure 3.7, the optimal choice for N_m is 2.

$\Delta_m > Lp_m$: It can be seen that Eq. (3.6) has no solution when $\Delta_m > L \cdot p_m$. However, the end-to-end delay requirements are not very stringent in this case. Although pipelining cannot be used, we still have message level parallelism for transmission on different links.

In practice, the choice of N_m is also affected by the size of the resulting packet group and the group delay. d_{grp} cannot be made very small because it has to include the transmission time of at least one packet, since packet transmission cannot be interrupted.

3.4.1 Channel Establishment

When a channel creation request is handled, we have to decide whether this request should be treated as a regular channel or a channel with packet groups. This decision is based on the parameters of the channel, Δ_m and p_m, and the number of links on the route to the destination. The channel establishment procedure has to be modified when packet groups are used, as shown in Figure 3.8. The decision on whether to consider the request as a channel with packet groups is taken in Step 2. The worst case delay for a link is again computed using procedure D_Order, but in this case, there is a target worst-case delay $d_m = \Delta_m/(N_m + L_m - 1)$ for the packet group on each link. The channel establishment check is that each link should be able to satisfy this target worst-case delay, without violating the worst-case delay of existing channels.

1. Select a source–destination route for the channel. This determines L_m, the number of links on the route.
2. Select N_m, the number of packet groups.
3. Compute the worst-case delay for the message on each link of the route. The desired worst-case delay is set to $d_m = \Delta_m/(N_m + L_m - 1)$.
4. Check the buffer requirement and compute the horizons for the different links.
5. Check whether the channel can be accommodated (Figure 3.9) for each link on the route. This is the channel establishment test.
6. If the channel can be established, assign d_m as the delay for each link on the route.

Figure 3.8: Channel establishment for packet groups.

Response Time Computation: The procedure D_Order also has to be modified to handle packet groups (see Figure 3.9). Since the channel to be established now has a target worst-case delay specified, it has to be assigned a priority based on this target d_m. This priority assignment can increase the worst-case response time, r_i', for existing channels, and may cause some of them to exceed the assigned worst-case delay, d_i. The new channel can be accepted only if $r_i' \leq d_i$ for all channels.

The response time computation scheme used in procedure D_Order now has to accommodate packet groups. Consider an existing channel which has response time r such that $i \cdot d_m < r < (i+1)d_m$, for some $i < N_m$. This channel can consider M_m to be a channel with period d_m and a message length equal to the size of the packet group. However, channels which have $N_m \cdot d_m < r$ will treat M_m as a regular channel with length C_m and inter-arrival time p_m.

Run-time Scheduling: Packets belonging to a packet group are considered as part of a single unit, and use the same logical arrival time. The logical arrival time of a packet group is determined on the basis of the logical arrival time of the message, and the sequence number of the packet group. For example, the logical arrival time for a packet belonging to the ith packet group of the message m_j is:

$$\ell t(packet) = i \cdot d_m + \ell(m_j)$$

1. Arrange the existing channels in ascending order of their associated delay d_i, where the delay is for a packet group of the channel.

2. Insert the new packet group into this order based on the required worst-case delay d_m.

3. Compute the new worst-case response times r_i' for all channels based on this priority assignment.

4. Check whether for all channels i, $r_i' \leq d_i$. If not, the channel cannot be established.

Figure 3.9: Extensions to Procedure D_Order for packet groups.

where $\ell(m_j)$ is defined as before for a regular channel. This computation has to be done only at the source node. At intermediate nodes, the logical arrival time is computed simply from the logical arrival time at the previous link. Here again, we can see how the use of a global time-base simplifies the design. Scheduling for the packet is handled as before based on this logical time and the horizon for the link.

Buffer Requirement: The intermediate nodes should provide buffer space sufficient to hold at least two packet groups. The buffer requirement is computed based on the horizon, as in Section 3.3, but with some small changes. The channel is considered to have an inter-arrival time of d_m and a message size equal to the size of the packet group, C_m/N_m.

3.5 Related Work

The concept of a unidirectional real-time channel was developed by the researchers of the DASH project. The real-time channel considered in this dissertation is similar to the deterministic real-time channel of [FV90]. That paper also considers two other types of channels: statistical, where messages are given a guarantee of successful delivery with a certain probability, and best-effort (with no guarantees). We do not consider statistical channels mainly because real-time applications require hard guarantees. Comer and Yavatkar have developed an abstraction called *flows* [CY88], which is similar to a unidirectional channel, but they do not give guarantees for the delivery time of messages. Flows can be identified with best-effort channels.

The parameters used for the description of the message arrival process are based

on the linear bounded arrival process model of Cruz [Cru87], with small modifications. Although the outline of our channel establishment scheme is similar to the one presented in [FV90], it is different in several respects. We have examined the problem of guarantee computation in depth, and developed a scheme for computing worst-case response times based on priorities. This corresponds to the delay bound test of [FV90]. It can be shown that the delay bound test of [FV90] is a special case of the priority-based response time computation scheme presented here. Moreover, we have presented an integrated solution to the problem of buffer reservation and flow control, based on the *horizon* model. We have also shown that channels with long messages require special consideration, and presented a scheme to accommodate them.

The problem of time-constrained communication in a multi-hop network has also been addressed by Cidon *et al.* in the PARIS system [CG88, CGGS88]. Their approach is based on providing limited buffer space in the switching nodes and using FIFO scheduling to obtain an upper bound on the network delay in each node. Limited buffering can result in packet loss, and so their efforts are aimed at maximizing throughput, while keeping the probability of packet loss below a certain level. Another approach to this problem is the "stop–and–go" queueing framework proposed by Golestani [Gol90a, Gol90b]. In this approach, a *framing* technique is used at all switching nodes to eliminate packet loss and to achieve a guaranteed end–to–end delay. However, the end–to–end delay is tied to the sizes of the frames, thus reducing the flexibility of satisfying different delay requirements. It appears that this approach is appropriate mainly when the required end–to–end delay is large compared to the periodicity of the message arrivals, as in voice transmission.

3.6 Summary

In this chapter, we have identified and solved problems related to the establishment and management of real-time communication channels. We have presented algorithms for computing the worst-case delay for messages, and for scheduling these messages. The computation was based on priority scheduling, and it was shown to be optimal under certain conditions. We also presented mechanisms for buffer allocation and flow control suitable for real-time channels, which would preserve the guarantees on delivery time. The ideas presented here will be integrated into the communication subsystem for HARTS, as described in Section 6.4.

APPENDIX

3.A Schedulability Analysis

Consider a set of channels $\{M_i = (C_i, d_i, p_i), i = 1, \ldots, m\}$ which share a common link ℓ, where C_i is the maximum service time on ℓ for any message on channel M_i, d_i is the maximum permissible delay for messages belonging to M_i at this link, and p_i is the inter-arrival time. Assume that the channel set is ordered by priority with M_1 being the channel with the highest priority. It can be shown that the worst-case delay for a message occurs when its arrival time coincides with the arrival time of all other messages of equal or higher priority. We can denote this critical instant as time $t = 0$. The transmission of a message can be delayed by the instances of higher priority messages present in the system at $t = 0$ and by subsequent arrivals of new instances of these messages. The arrival times of these instances are given by the set S_i, and the system time requirement of this message and of the higher priority messages is given by $W_i(t)$, as shown below (see [LSD89]).

$$S_i = \{d_i\} \bigcup \{kp_j \mid j = 1, 2, \ldots, i-1; k = 1, 2, \ldots, \lfloor (d_i/p_j) \rfloor\}$$

$$W_i(t) = \sum_{j=1}^{i-1} C_j \cdot \lceil t/p_j \rceil + C_i \tag{3.7}$$

A channel M_i is schedulable (i.e., its delay bound will always be satisfied) *if and only if* $W_i(t) \leq t$, for some $t \in S_i$. The worst-case response time for messages belonging to M_i is the smallest value of t such that $W_i(t) = t$. The set of channels is said to be schedulable if and only if each channel in the set is schedulable.

These results hold under the conditions that preemptive scheduling is employed and a message can be aborted at any stage in its execution without any loss. In our model, messages are split into packets, and packet transmission cannot be interrupted. The schedulability conditions have to be modified to account for the new worst-case conditions. The worst-case response time for a message occurs when all messages of equal and higher priority arrive at the same time and a packet belonging to a lower priority message is being transmitted at that time. The system time requirement in this case is:

$$W_i'(t) = C_{pkt} + \sum_{j=1}^{i-1} C_j' \cdot \lceil t/p_j \rceil + C_i', \tag{3.8}$$

where C_{pkt} is the service time for the maximum size packet, and C_j' is the service time for a message on M_j, including the overheads of packetization.

CHAPTER 4

THE ROUTE SELECTION PROBLEM

4.1 Introduction

The problem addressed in this chapter is the selection of routes for messages in a multicomputer system of this type. It is assumed that the assignment of tasks to different nodes has already been done and the communication patterns between tasks are known. The reason for this is that, in a real-time system, the placement of tasks depends not only on the communication pattern but also on the peripherals and resources available at the different nodes. The traffic between nodes is specified in terms of mean data rates, which can be determined from the length of the messages and the periodicity of the communication. Given the network topology and this traffic pattern, the objective is to select a route for each pair of communicating processes such that the network load is balanced and the probability of establishing virtual cut-through is enhanced. Route selection is essential in order to reduce message delivery delays and to make full use of the network's cut-through capabilities. It can be considered to be a high-level approach to message scheduling, where the units considered are inter-process traffic patterns and not individual packets.

In the type of network considered here, it is assumed that link capacities are large and the overheads in handling packets at intermediate nodes is a significant factor. This assumption is reasonable considering the Gigabit range bandwidth possible with fiber-optic links. Also, we restrict our attention to finding a single fixed route for each pair of communicating processes. Fixed routing is preferable in real-time systems because it can simplify the computation of bounds for the message delivery time, an issue which is important when messages with delivery time constraints are considered [Fer89].

Variations of this problem have been studied for general wide-area networks, mainly as techniques for building routing tables in packet switched networks [SS80]. However, most of these studies are based on link capacities, since these are limited for wide-area networks. In the TRANSPAC network, the routing algorithm uses link costs depending

upon utilization [SS80]. The cost function is piecewise continuous with hysteresis. The routing is computed to minimize costs depending upon the current utilization or queue lengths, but future traffic is not considered.

One study relating to multiprocessor networks is by Bianchini and Shen [BS87], in which a traffic scheduling algorithm is presented. However, they assume that the switching nodes in the network have the ability to implement traffic splitting, which makes the network behave like a fluid-flow pipeline. They further assume that the delivery cost is not depended on the length of the path and arrive at a link cost which is an exponential function of the link traffic. The properties of the exponential link cost function result in a major simplification for their traffic scheduling algorithm. On the other hand, the message delivery delays in a multicomputer network have a strong dependence on the length of the path and this is taken into account in the cost function derived here.

In this chapter, the traffic scheduling problem is formulated as an optimization problem in Section 4.2. The link cost function used in this formulation is derived in Section 4.3. In Section 4.4 it is shown that the optimization problem is $\mathcal{NP}$-Hard and the associated decision problem is $\mathcal{NP}$-Complete. The heuristic algorithms developed to solve this problem and their basis are presented in Section 4.5. The performance of these algorithms is then analyzed using simulation in Section 4.6. The chapter concludes with Section 4.7.

4.2 Notation and Problem Formulation

The environment under consideration is a multicomputer system with a point-to-point interconnection structure and a set of assigned tasks. It is assumed that the message communication patterns between the tasks are known in terms of average length of messages and the frequency of communication. From this, one can determine the unidirectional task-to-task communication volume. We call this volume a *flow*, which is measured in bytes/sec. The problem is then to determine a route through the network for each flow to meet a global "goodness" criterion.

The notation used to describe this problem formally is given in Table 4.1. For brevity, the term *path* is used to denote a *simple directed path* in the digraph DG. Also, the terms *path* and *route* are used interchangeably in this chapter. Given these definitions, and the condition that there is a unique path for each flow, the routing problem can be stated as follows.

Path Selection Problem: Given N, L, Q, the capacities and the costs of the links,

DG: A digraph (N, L) representing the multicomputer system which consists of a set, N, of n nodes and a set, L, of m links.

e_{ij}: The directed link/edge from node i to node j.

Q: The set of q flows $\{(s_i, d_i, r_i)$: flow r_i required from node s_i to node d_i in DG, $i = 1, \ldots, q\}$.

P_k: The set of all simple directed paths from s_k to d_k in DG, where $(s_k, d_k, r_k) \in Q$.

p_k: An element of P_k, which can be considered to be an ordered subset of L.

S: $\{p_k : k = 1, \ldots, q\}$, an element of $P_1 \times P_2 \cdots \times P_q$.

f_{ij}: The flow in link e_{ij}. $f_{ij} = \sum_{k=1}^{q} \mathrm{I}_{ij}^k r_k$, where $\mathrm{I}_{ij}^k = 1$ if $e_{ij} \in p_k$ and 0 otherwise.

C_{ij}: The capacity of link e_{ij}.

c_{ij}: The cost of routing traffic onto link e_{ij}, which can be a function of f_{ij} and/or C_{ij}.

Table 4.1: Notation

find a set, S, of paths such that for each link $e_{ij} \in L$, $f_{ij} \leq C_{ij}$ and total cost T = $\sum_{k=1}^{q} r_k \cdot (\sum_{e_{ij} \in p_k} c_{ij})$ is minimized.

By changing the order of summation, T can be rewritten as T = $\sum_{e_{ij} \in L} c_{ij} \cdot (\sum_{k=1}^{q} I_{ij}^{k} r_k)$. The inner summation is seen to be the link flow f_{ij}, so T = $\sum_{e_{ij} \in L} c_{ij} \cdot f_{ij}$.

As formulated above, the objective of route selection is to minimize the total cost over all the flows in the network, while adhering to the capacity constraints for each link. Although this formulation is general, the characteristics of a particular type of network can be incorporated by a suitable choice of the link cost function c_{ij}. The features of virtual cut-through switching make the flow through a link critical to the cost function, since it affects the probability of establishing a cut-through route. Intuitively, it would be better to choose longer than optimal routes so as to avoid heavily used links. However, path lengths remain an important factor because long paths use up more network resources. It will be shown in the next section that the link cost function can be chosen such that minimizing the total cost is equivalent to maximizing the weighted probability of achieving virtual cut-through in the network. The total cost function T then represents a tradeoff between using longer than optimal paths to avoid busy links versus the penalty incurred by the increased path length.

4.3 Derivation of the Link Cost Function

It is our intention to minimize the probability of messages getting buffered at intermediate nodes in the path. When a message gets buffered, it has to be examined by the processor on the communication adapter and then scheduled for onward transmission. Thus, in addition to the message store-and-forward delay, an additional processing cost is incurred. Hence, it is preferable to use a long path with a lower probability of buffering. The link cost function is derived based on this probability.

The network $DG = (N, L)$ is modeled as a network of queues with one (single server) queue for each link. The problem of analyzing a network of queues with arbitrary arrival and service patterns is known to be intractable. To make the analysis tractable, the following assumptions are made in this analysis. It is noted that these assumptions may not be valid for a real-time system where periodic message traffic is predominant. However, their use is justified, since the results obtained are used mainly as a guideline for the choice of the link cost function.

A0: Poisson message generation at the source nodes.

A1: Exponentially distributed message lengths.

A2: Infinite nodal capacity.

A3: A packet loses its identity when it arrives at a node and a new length is chosen for it at random, i.e., the Independence Assumption [Kle64].

This model, and the Independence Assumption, was first used by Kleinrock [Kle64], and simulations and actual measurements have later demonstrated the validity of the model. The model has also been used by other researchers for the analysis of networks [KK79, IM86]. Under these assumptions, the arrival process for a link is independent of the departure process from the link and Jackson's result [Jac57] can be applied to the network of queues. That is, in the steady state the network behaves as if each node were stochastically independent of the other nodes and similar to an M/M/1 system.

Consequently, the probability that a message gets buffered at a particular intermediate node is independent of the probability of it getting buffered at any other intermediate node. Hence, the probability of establishing a source–destination cut-through route can be written as a product form expression. Consider a path $P = \{e_{n_0 n_1}, e_{n_1 n_2}, \ldots, e_{n_{\ell-1} n_\ell}\}$ from node $n_0 = i$ to node $n_\ell = j$. The probability that a message from node i to node j will be delivered without buffering at intermediate nodes is given by:

$$\begin{aligned} Prob(\text{cut-through on } P) &= \prod_{r=1}^{\ell-1}(1 - Prob(\text{buffering at node } n_r)) \\ &= \prod_{r=1}^{\ell-1}(1 - Prob(\text{link } e_{n_r n_{r+1}} \text{ is busy})) \\ &= \prod_{r=1}^{\ell-1}(1 - \rho_{n_r n_{r+1}}) \\ &= 1 - \sum_{r=1}^{\ell-1} \rho_{n_r n_{r+1}} + \text{higher order terms in } \rho \end{aligned}$$

where $\rho_{n_r n_{r+1}} = f_{n_r n_{r+1}} / C_{n_r n_{r+1}}$ is the mean utilization of link $e_{n_r n_{r+1}}$.

In the type of high-bandwidth network considered here, the mean utilization of the links would tend to be small. Hence, to a good approximation, the higher order terms can be dropped from the equation. Therefore,

$$Prob(\text{cut-through on } P) \approx 1 - \sum_{r=1}^{\ell-1} f_{n_r n_{r+1}} / C_{n_r n_{r+1}}.$$

$Prob(\text{cut-through on } P)$ is maximum when $\sum_{r=1}^{\ell-1} f_{n_r n_{r+1}} / C_{n_r n_{r+1}}$ is minimum.

To consider all flows in Q we have to form a weighted sum over all q flows and maximize this:

$$\sum_{k=1}^{q} r_k \cdot Prob(\text{cut-through on } p_k).$$

In this sum, the utilization of the first link in the path will not be included because the expression for cut-through on p_k includes only intermediate nodes. However, for this analysis, an approximation is used in which the utilization of the first link is also considered, since a lower utilization for the first link would ensure that the delay experienced before the message is transmitted from the source node is small. This then corresponds to finding the minimum of $\sum_{k=1}^{q} r_k \cdot (\sum_{e_{ij} \in p_k} (f_{ij}/C_{ij}))$.

The form of this function matches the cost function of the Path Selection Problem, with the link cost $c_{ij} \equiv f_{ij}/C_{ij}$. If we restrict our attention to homogeneous networks, the capacity C_{ij} is the same for all links and the cost function can be simplified to $c_{ij} = f_{ij}$. This function can be interpreted as follows. As the number of flows through a link increases, f_{ij} increases and so does the cost of using the link. This would present a bias towards the choice of longer paths to circumvent heavily used links. On the other hand, the total cost for a flow depends upon the length of the path and so, there is an opposing bias to reduce the length of the paths. With $c_{ij} = f_{ij}$ as the link cost function, the total cost $T = \sum_{e_{ij} \in L} f_{ij}^2$. This is the form of the total cost function used in subsequent sections.

4.4 Problem Characterization

In this section it is shown that the Path Selection Problem is $\mathcal{NP}$-Hard and the decision problem associated with it is $\mathcal{NP}$-Complete. The decision problem, shown below as Decision Problem 1, is $\mathcal{NP}$-Complete because it contains the subproblem of finding a feasible set of paths. It can be shown that SATISFIABILITY (SAT) [Coo71] is reducible to this subproblem of finding feasible paths. The reduction relies on capacity constraints which force exclusive use of links. However, even when the capacity constraints are not imposed, the problem remains $\mathcal{NP}$-Complete when the link cost c_{ij} is a function of the link utilization. In particular, it is shown that Decision Problem 2 given below, which has the cost function $c_{ij} = f_{ij}$ and no link capacity constraints, is $\mathcal{NP}$-Complete. The definition of SAT, adapted from [GJ79], is given in Appendix 4.A.

Decision Problem 1 *Given N, L, Q, and the link capacities C_{ij}. Is there a feasible set, S, of paths such that for each edge $e_{ij} \in L$, $f_{ij} \leq C_{ij}$ where $f_{ij} = \sum_{e_{ij} \in p_k} r_k$?*

Decision Problem 2 *Given N, L, Q, link cost function $c_{ij} = f_{ij}$, and a bound B. Is there a feasible set, S, of paths such that total cost* $= \sum_{k=1}^{k=q} r_k \cdot (\sum_{e_{ij} \in p_k} c_{ij}) = \sum_{e_{ij} \in L} f_{ij}^2 \leq B$?

The reduction from SAT is similar for the two problems and it is shown only for the second problem. It is based on the method used in [EIS76] for the Multicommodity Integral Flow problem. The proof is constructive and it shows a polynomial time transformation from an instance of SAT to an instance of Decision Problem 2.

Theorem 3 *Decision Problem 2 (DP2) is $\mathcal{NP}$-Complete.*

Proof: It is easy to see that DP2 is in $\mathcal{NP}$ because given a guess for the set of paths, it is possible to compute and verify that the total cost is within the bound B, in polynomial time. We now show that SAT is reducible to DP2.

Given an instance of SAT with collection $C = \{C_1, C_2, \ldots, C_\ell\}$ of clauses on a finite set U of variables. For each variable x_i, let t_i be the number of occurrences of x_i in the clauses, and u_i be the number of occurrences of $\overline{x_i}$. Let $w_i = \max(t_i, u_i)$. Construct a graph component G_i corresponding to this variable as shown in Figure 4.1. The component consists of a start node v_s^i, an end node v_t^i, and two chains of nodes, each with $2w_i$ nodes. The chains represent a choice of truth assignment for the variable. These components are then connected in series with the end node of component G_i connected to the start node of component G_{i+1}, that is, v_t^i connected to v_s^{i+1}. Create special nodes s and d and links from s to v_s^1, $v_t^{|U|}$ to d.

For each clause C_i create two nodes, s_i and d_i. For the j-th occurrence of literal x_i, say in clause C_ℓ, create links from from s_ℓ to v_{2j-1}^i and from v_{2j}^i to d_ℓ. A similar construction is used for an occurrence of $\overline{x_i}$, but in that case, the nodes $\overline{v_{2j-1}^i}$ and $\overline{v_{2j}^i}$ would be used. The graph components and the additional links described above, together form the sets N and L. The set of flows is chosen to be $Q = \{(s, d, 1), (s_1, d_1, 1), \ldots, (s_{|C|}, d_{|C|}, 1)\}$. Finally, the bound B is selected to be $3|C| + 1 + \sum_{i=1}^{|U|}(2w_i + 2)$. This bound is in fact the total cost of routing the set of flows Q, where each link has *exactly one* flow. The $3|C|$ part corresponds to the flows associated with the clauses, and the $(1 + \sum_{i=1}^{|U|}(2w_i + 2))$ part corresponds to the $(s, d, 1)$ flow. This gives an instance of DP2.

If the instance of SAT has a feasible solution, there is a truth assignment to the variables which satisfies all the clauses. In the corresponding instance of DP2, the following solution is feasible. The path from s to d is chosen such that if x_i is assigned a FALSE value, the upper trail $(v_1^i, v_2^i, \ldots)$ is chosen through the component G_i, and vice versa. In

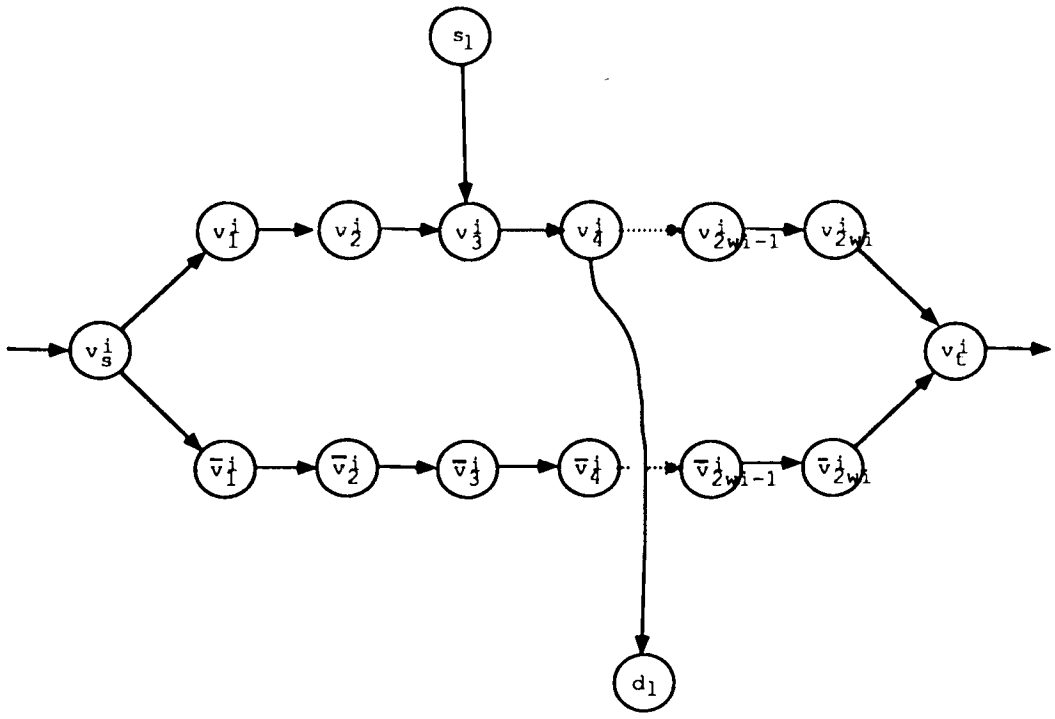

Figure 4.1: Graph component corresponding to a variable.

each clause C_j, there exists a term u_k^j which evaluates to TRUE and the link in the graph component corresponding to this term, say e_j, would be unused. The path for (s_j, d_j) can be chosen to go through this link. It can be verified that this choice of paths gives a feasible solution for DP2.

Similarly, given a solution to DP2, the path from s to d through the component G_i determines the truth assignment for the corresponding variable. The choice of B forces the paths selected to be disjoint. Otherwise, if a link were used in more than one path, its cost will be greater than 1 and the total cost will exceed B. This ensures that a variable cannot be used with conflicting values in different clauses, and the path (s_j, d_j) identifies a term in C_j which has a TRUE value. Thus, it is clear that the instance of SAT has a feasible truth assignment *iff* there is a feasible set of paths for the instance of DP2. ■

Corollary 1 *The Path Selection Problem with cost function $c_{ij} = f_{ij}$, and no capacity constraints, is $\mathcal{NP}$-Hard.*

The Path Selection Problem is a search problem which has a finite upper bound, so it can be easily shown that it is Turing reducible to DP2 using the "binary search" technique described in [GJ79]. Hence, it follows from the theorem that the Path Selection

problem is $\mathcal{NP}$-Hard. Also, from the proof of Theorem 3 it can be seen that a bound, B, can be suitably chosen to accommodate other cost functions where the link cost is a monotone increasing function of the link utilization.

4.5 Solution Algorithm

The problem of finding a minimum cost solution has been shown to be $\mathcal{NP}$-Hard when the link cost function is a function of the utilization. It is observed that the search space grows as $\prod_{k=1}^{q} |P_k|$, where $|P_k|$ is usually large for the types of regular interconnection networks that are under consideration. This implies that it is impractical to determine the optimal solution by brute-force techniques, even for a small number of flows. Thus, we need to find a good heuristic solution for the problem.

We have developed an algorithm which selects paths one at a time, while keeping the other paths fixed. The algorithm begins with an initial assignment of paths to the flows. It then successively tries to improve the paths, considering one flow at a time, until no further cost improvement is possible. An outline of this algorithm, which is called Route_All, is given in Figure 4.3. The procedure for determining the path for a single flow, called Route_One, is described in Figure 4.2. The theoretical basis for this procedure is given below.

The algorithm required for selection of a single route is one in which we are given the current utilization of the network and the characteristics of the new flow to be established. The objective here is to find a route such that, under the given cost function, the incremental cost is minimized. It is shown that a version of the *shortest path* algorithm with appropriately defined link lengths will yield a solution.

Consider a path p_k for this new flow. When this flow is added to the system, the new flows in the links are given by

$$f'_{ij} = \begin{cases} f_{ij} + r_k & \text{if } e_{ij} \in p_k \\ f_{ij} & \text{otherwise.} \end{cases}$$

The additional cost incurred is then

$$\begin{aligned} I &= \sum_{e_{ij} \in L} [(f'_{ij})^2 - f_{ij}^2] \\ &= \sum_{e_{ij} \in p_k} (2 f_{ij} r_k + r_k^2) \\ &= r_k \cdot \sum_{e_{ij} \in p_k} (2 f_{ij} + r_k). \end{aligned}$$

Algorithm Route_One

```
/*
    Given N, L, F={f_ij : e_ij ∈ L} and a new flow (s_k,d_k,r_k),
    find a route p_k with minimum incremental cost.
*/

    /* Initialize the distance matrix D_ij */
    for each i,j ∈ N
        if e_ij ∈ L
            D_ij := 2 * f_ij + r_k
        else
            D_ij := 0
        end if
    end for

    Use the Greedy Algorithm to find a path p_k from s_k to d_k
    with the distance matrix D_ij.

end Route_One
```

Figure 4.2: Single path selection.

This incremental cost, I, is minimum when $\sum_{e_{ij} \in p_k}(2f_{ij} + r_k)$ is minimum. This minimum can be obtained by using $2f_{ij} + r_k$ as the *length* of link e_{ij} and finding the shortest path from s_k to d_k. This is the approach used in Algorithm Route_One, in which a greedy algorithm is used to find the shortest path. Since all the link lengths are non-negative, a greedy algorithm would yield the shortest path. From this discussion, it is clear that Algorithm Route_One is optimal under the condition that routes that are already established cannot be disturbed. This is the case when requests for creation of new flows arrive dynamically and are serviced in order.

Algorithm Route_All starts with an initial flow assignment, F. It then selects each flow in turn, removes it from F, and checks whether a better route is available using Route_One. The function *Incremental_cost* computes I as shown in the derivation above. The solution obtained is not guaranteed to be optimal because it is possible that a better solution may be obtained by the simultaneous re-routing of multiple paths. The algorithm, however, is guaranteed to terminate since the cost function is monotone decreasing with successive re-routing. Algorithm Route_One is essentially a shortest path algorithm and its complexity is $O(N^2)$ for an adjacency matrix representation of the network. Algorithm Route_All uses Route_One repeatedly, but the number of iterations is data dependent and cannot be easily determined. The running time of Route_All is bounded below by $q \cdot N^2$, that is, it is $\Omega(qN^2)$. However, the results of experiments with the algorithm on several

Algorithm Route_All

```
/*
    Given N, L, and Q, find a set of paths for each flow in Q
    with the objective of minimizing the cost function.
*/

    Assign an initial path for each flow, initialize the flow matrix F.

    new_route_found := TRUE
    while (new_route_found)
        new_route_found := FALSE

        for each (s_i, d_i, r_i) in Q
            remove p_i from F
            old_cost = Incremental_cost(p_i)

            Route_One (s_i, d_i, r_i) returns path p'_i

            new_cost = Incremental_cost(p'_i)
            if (new_cost < old_cost)
                add p'_i to F
                new_route_found := TRUE
            else
                restore p_i to F
            end if
        end for

    end while

end Route_All
```

Figure 4.3: The route selection algorithm.

flow patterns show that the convergence is rapid.

4.6 Performance Evaluation

In this section, we evaluate the performance of the algorithms developed in Section 4.5, and study the effectiveness of the cost function derived in Section 4.3. To evaluate the performance of Algorithm Route_All, termed ALLP, it is compared with two other path selection algorithms. One of them, called INC, is a simple extension of Algorithm Route_One in which routes are assigned to the flows in the order of arrival and no further re-routing is employed. The other is a "pure" shortest path algorithm (SP) in which a path of minimum length from source to destination is selected for each flow. The purpose of this comparison is to measure the improvement obtained using the re-routing technique. The comparison between SP and INC demonstrates the utility of Algorithm Route_One as compared to a simple shortest path algorithm. The complexity of INC is the same as that of the shortest path algorithm, $O(N^2)$ for an adjacency matrix representation of the network. The execution time is slightly higher because it has to update the flow f_{ij} in the links which constitute the route after the route has been selected.

To evaluate the effectiveness of the cost function, we used a discrete-event simulator which models a network with virtual cut-through switching. This simulator was originally developed by the authors of [DRS91] to study the behavior of virtual cut-through in HARTS and it models the routing hardware, its interface to the buffer management unit, and the network processor of each node.

The simulator accurately models the delivery of each message by emulating the routing hardware along the route of a packet at the microcode level. It also captures the internal bus access overheads experienced by packets as they pass through an intermediate node. For example, when a transit packet arrives at an intermediate node, the following sequence of events is initiated. First, the receiver for the link on which the packet arrived waits for the packet header to become available. It then examines the packet header to determine the packet type. For a BROADCAST packet, the receiver tries to schedule two events: one to reserve the transmitter in the same direction to forward the packet, the other to the buffer management unit to receive the packet. Lastly, the receiver schedules events to signal the completion of the packet at this node. The simulator collects detailed statistics for different types of messages. In addition to supporting exponentially distributed packet lengths, the simulator can also use a discrete distribution of packet lengths in which the user specifies different types of messages, their lengths, and the probability of generation of

each type of message.

We modified the simulator to generate traffic based on each connection (flow), and to collect statistics on the number of *bufferings*, i.e., the number of times packets failed to cut-through and were buffered at intermediate nodes. We chose to measure the number of bufferings because that is what the cost function tries to capture. We did not use the average delivery time as a measure because the overhead cost incurred when a packet is buffered depends upon the processor, and the load on the processor. The simulator did not model this load.

The simulator allows us to relax many of the modeling assumptions used in the derivation of the cost function. In particular, we do not use the Independence assumption, nor do we have exponentially distributed message lengths. In our experiments, we configured the simulator for a hexagonal mesh of size 5 (denoted by E-5), which has 61 nodes. We generated sets of flows by selecting the source, destination, and quantity of each flow using independent random number generators. The source node was chosen using a uniform random number generator. For a fixed source, the destination node was chosen using an independent random number generator, with two different types of distributions in different experiments. One was a uniform distribution, where the destination node is chosen to be any other node in the system with equal probability. The second distribution tried to capture the principle of locality, that is, the destination node is more likely to be close to the source. The value of each flow was selected in the range [1,10] using a third uniform random number generator. Note that in the cost function, it is the *relative* values of the flows which are significant, the units do not affect the results.

We used the SP, INC, and ALLP algorithms to produce 3 different sets of route assignments for each set of flows. These algorithms also returned the "cost" of the route assignments that they produced. We then used the mesh simulator to simulate traffic on these flows and measured the number of bufferings for each of the three sets of routes. The value of a flow was interpreted to be a rate of packet generation, and the packet size was fixed at 128 bytes. On each flow, packets were generated using a Poisson distribution with a mean inter-arrival time specified by the value of the flow. The actual rate of packet generation and the service time for the packets are a function of the link transmission rate. The effective link transmission rate in our simulation was 1.5 microseconds per byte, and a flow value of 1 was converted into an arrival rate of 1 packet per 180 milliseconds. This value was chosen to ensure that the average link utilization remained low (it was below 0.4 in all our experiments). We note that the number of bufferings observed is a function of link

utilization and it would change if we assigned a different rate for the flow value. We could make this arbitrary choice of generation rate because, in our experiments, we are interested in the relative values of the number of bufferings for the three route assignments. In each simulation, we collected statistics over a period in which 100,000 packets were delivered. This number was chosen by looking at the convergence of the statistic of interest (number of bufferings) in some sample simulations.

Experiment 1: In this experiment, for any source, the destination of the flow was chosen using a uniform distribution. The number of flows was varied from 50 to 500, and this resulted in a change in the observed average link utilization from 0.035 to 0.33. The results of the simulation are shown in Figure 4.4, which plots the number of bufferings observed for a particular number of flows with each of the three algorithms. Each data point in the graph represents an average over 50 different data sets. The results here can be compared with the cost figures shown in Figure 4.5 for the same sets of flows. This experiment also allows us to compare the performance of the three algorithms. The comparison shows that the computed cost and the observed number of bufferings have similar behavior, except when the number of flows is very small. In this case, INC and SP perform better than what the cost function shows. It is seen that there is a modest improvement for INC over SP, and for ALLP over INC. The improvement for INC over SP is higher when there are fewer flows, that is, when the network load is light, because INC is able to locate routes through links that are otherwise unused. Since INC performs quite well in this situation, the additional improvement attainable by using ALLP is small. On the other hand, when the number of flows is very large, because of the uniform communication pattern, most of the links are almost evenly loaded and INC is not able to find much improvement.

Experiment 2: This experiment differs from Experiment 1 in that the destination node was picked by first selecting a hexagonal ring and then selecting the particular node within the ring, considering the source node as the center of the mesh. Since there are more nodes in the outer rings, the probability of selecting a node in an outer hexagon is smaller than that of selecting a node from an inner hexagon. With this distribution, the average source–destination separation in a mesh of size e is $e/2$ as compared to $(2e-1)/3$ in the case of a uniform distribution. When the number of flows was varied from 50 to 400, the observed average link utilization changed from 0.035 to 0.26. The number of flows could not be increased beyond 400 for the simulation because the routes generated by SP resulted in severe network congestion. The results of the simulation are shown in Figure 4.6,

while those of the cost function are shown in Figure 4.7. Here again, the results for the simulation match well with those given by the cost function. There is an apparent anomaly seen here, in that, the cost for the SP algorithm is vastly increased even though the average distance for a route is reduced as compared to the uniform distribution. The reason for this is that, although the route length is longer in Experiment 1, the number of different paths to nodes in an outer hexagon is substantially larger. Hence, because of the uniform distributions for sources and destinations, the SP algorithm performed quite well. However, for the non-uniform distribution, where the average number of possible shortest paths for a flow is smaller, the effects of congestion are more pronounced and so the performance for SP deteriorates. The INC and ALLP algorithms are better equipped to cope with congestion and this is reflected in their vastly better performance than SP.

From these experiments, we can see that a route assignment which reduces the cost function also results in a reduction of the number of bufferings in a hexagonal mesh network.

Other Experiments

To evaluate the performance of the routing algorithms, we have to test them on other topologies and other mesh sizes. However, the computing resources required for comparing the performance using the number of bufferings is prohibitive. The mesh simulator ran on a SUN Sparcstation 1, and the time required for the simulation of a single set of routes was several minutes. Consequently, we evaluate the algorithms based only on the cost function. We have tested the routing algorithms on two network topologies: binary hypercubes and hexagonal meshes, and for different network sizes. This choice was motivated by the fact that variations of virtual cut-through switching have been implemented for these topologies. The binary hypercube is a well-studied topology [Sei85] and is used in many commercially available multicomputer systems, hence its description is omitted.

The performance of these algorithms has been studied using simulated traffic patterns which were generated as described earlier. For each data point, the experiment was repeated with 100 different sets of flows and an average value was obtained. The standard deviation of the mean was found to be less than 5% of the mean for all data points, and less than 3% for most data points.

Figures 4.8 and 4.9 show the results of the simulation on a hexagonal mesh of size 4 (denoted by E-4), having 37 nodes, using a uniform distribution for the selection of the destination node. A comparison of the total cost for the solutions obtained by the

three algorithms is shown in Figure 4.8, and the percent improvement in cost between SP and INC, and between INC and ALLP, is shown in Figure 4.9. These results are similar to those obtained for the E-5 mesh in Experiment 1. In the second set of experiments for the hexagonal mesh, the destination node was selected using a non-uniform distribution as in Experiment 2. The simulation results obtained with this non-uniform distribution for an E-4 mesh are shown in Figures 4.10 and 4.11, and they are similar to the results obtained for the E-5 mesh in Experiment 2. We also performed experiments on an E-6 mesh and the results obtained were similar.

The experiments with the binary hypercube used a uniform distribution for the selection of source and destination nodes, but in this case, the average distance between source and destination is $n/2$ for a hypercube of dimension n. The results obtained for hypercubes of dimension 5 (Q-5, 32 nodes) and 6 (Q-6, 64 nodes) are shown in Figures 4.12 to 4.15. These are similar to the results for the hexagonal mesh with a non-uniform source-destination distribution. They also show that the INC and ALLP algorithms perform much better than the SP algorithm. ALLP again shows a modest improvement over INC.

In Algorithm Route_All, the final results can depend on the initial state of the network. To study the sensitivity of the results to the initial flow assignment, three different strategies were used to select this assignment. The first method was to select a route using the shortest path algorithm without considering the other routes in the network. The second method used the Route_One algorithm, with routes chosen in the order in which the flows were generated. In the third strategy, the flows were sorted in the order of cost before using the Route_One algorithm for assignment. Both the ascending and the descending order of cost were considered. A comparison of these strategies was made for an E-4 mesh using the same set of flows as input, and the experiment was repeated several times with different sets of flows. The results indicated that, on the average, there was very little difference in the final cost.

4.7 Summary

The problem of routing inter-processor message traffic in a point-to-point interconnection network has been formulated as an optimization problem for the total cost, where the cost of a route depends upon the links that are used. The link cost function $c_{ij} = f_{ij}$ was chosen with the objective of maximizing the probability of establishing virtual cut-through routes in the network using analysis based on a queueing model for the network. It is noted that although some of the assumptions made in the queueing analysis

may not be valid in a real-time system, the cost function obtained intuitively captures the notion of congestion avoidance. This was verified in our experiments with the hexagonal mesh simulator.

The optimization problem was shown to be $\mathcal{NP}$-Hard, and thus, a heuristic algorithm (ALLP) was developed for the solution. The special case of online route selection was also considered, in which requests are serviced as they arrive, and a polynomial time algorithm (INC) was developed for this case. The performance of these algorithms was studied for the binary hypercube and the hexagonal mesh network topologies using simulation. It was found that the extent of the performance improvement depends upon the network topology and the distribution of the source–destination pairs in the network.

The ALLP algorithm can be used for selecting routes off-line in conjunction with an algorithm for assignment of tasks to the nodes. The cost function of ALLP can then be incorporated into the cost of the task assignment to reflect the cost of communication. On the other hand, the INC algorithm can be used to select routes for flows in a dynamic environment. It is optimal for the cost function derived here under the condition that routes that are already established cannot be disturbed. This algorithm has been applied to select routes for real-time channels, which is the first step in the channel establishment procedure of Section 3.2.

APPENDIX

4.A The Satisfiability Problem

Let $U = \{x_1, x_2, \ldots, x_m\}$ be a set of Boolean variables. If u is a variable in U, then u and $\overline{u}$ are *literals* over U. A truth assignment for U is a function $t : U \rightarrow \{T, F\}$, which corresponds to *true* and *false* respectively. The literal u is true under t if and only if the variable u is true under t. Similarly, the literal $\overline{u}$ is true if and only if the variable u is false.

A *clause* over U is a set of literals over U, and represents the disjunction of those literals. It is *satisfied* by a truth assignment if and only if at least one of its members is true under that assignment. A collection C of clauses over U is *satisfiable* if and only if there exists some truth assignment for U that simultaneously satisfies all the clauses in C. With this definitions, the SATISFIABILITY problem can be stated as follows. It was shown to be $\mathcal{NP}$-Complete by Cook [Coo71].

SATISFIABILITY Given a set U of variables and a collection C of clauses over U. Is there a truth assignment for U such that C is satisfiable?

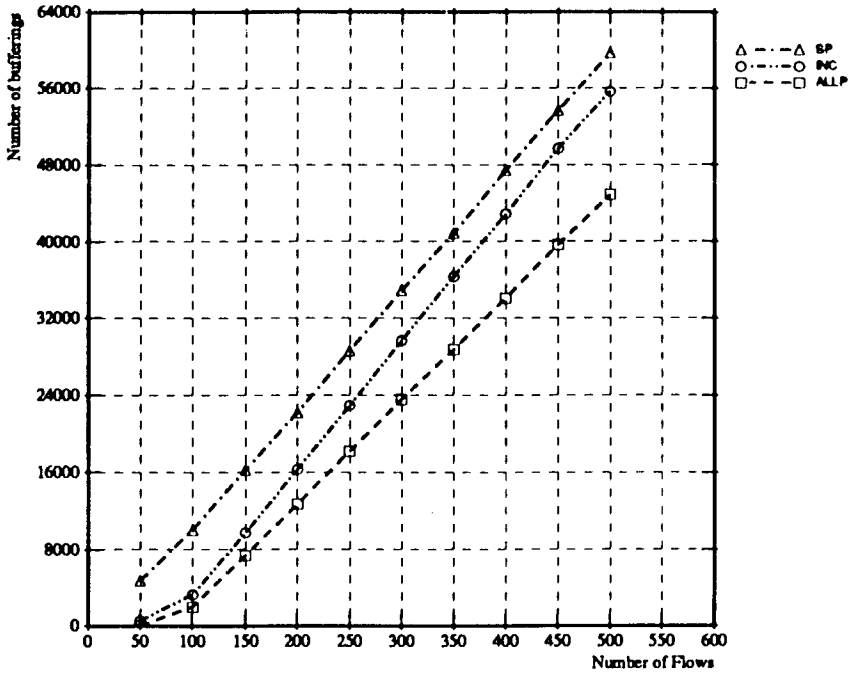

Figure 4.4: Comparison using number of bufferings: E-5 mesh, uniform distribution.

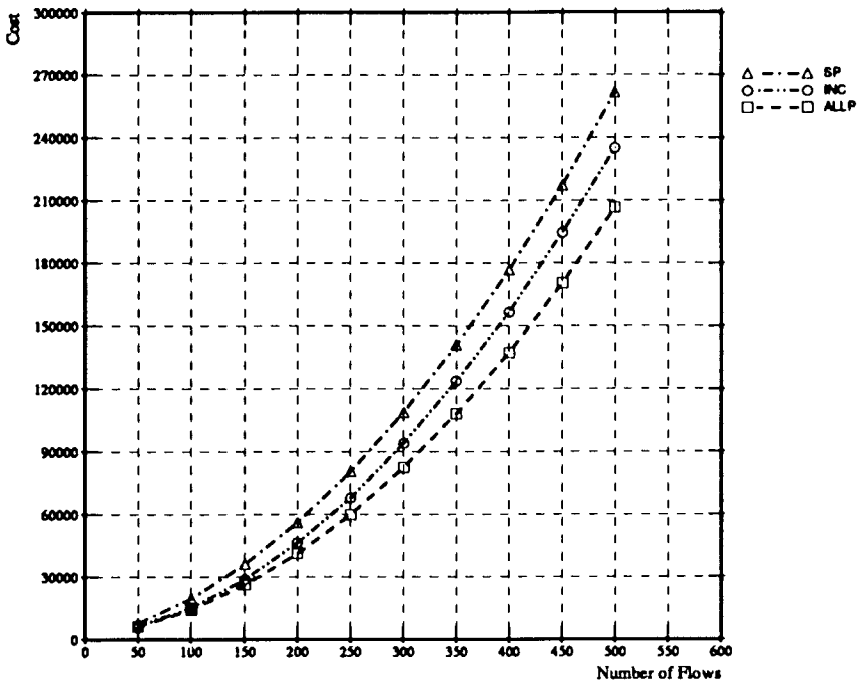

Figure 4.5: Comparison using the cost function: E-5 mesh, uniform distribution.

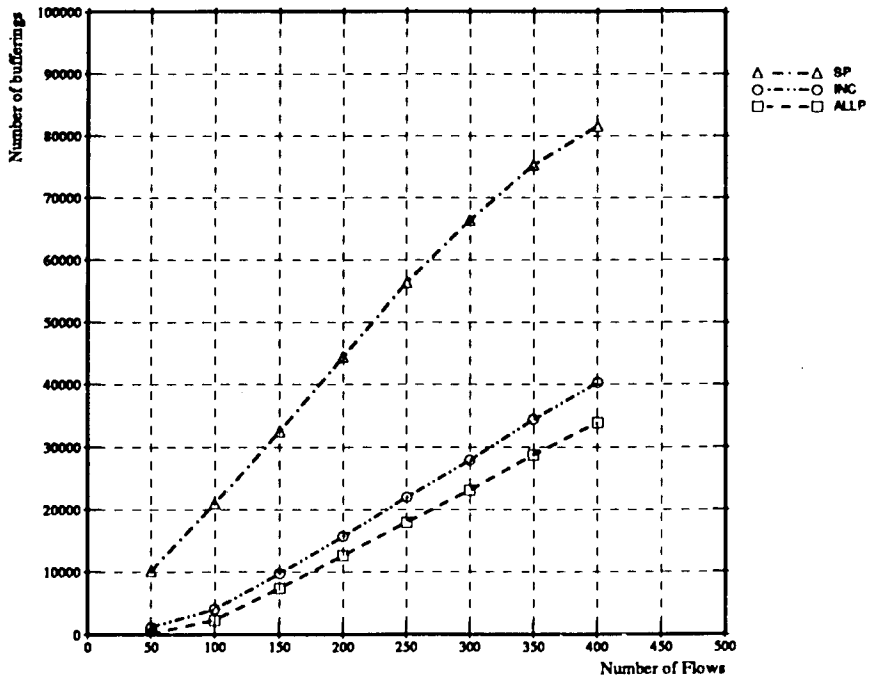

Figure 4.6: Comparison using number of bufferings: E-5 mesh, non-uniform distribution.

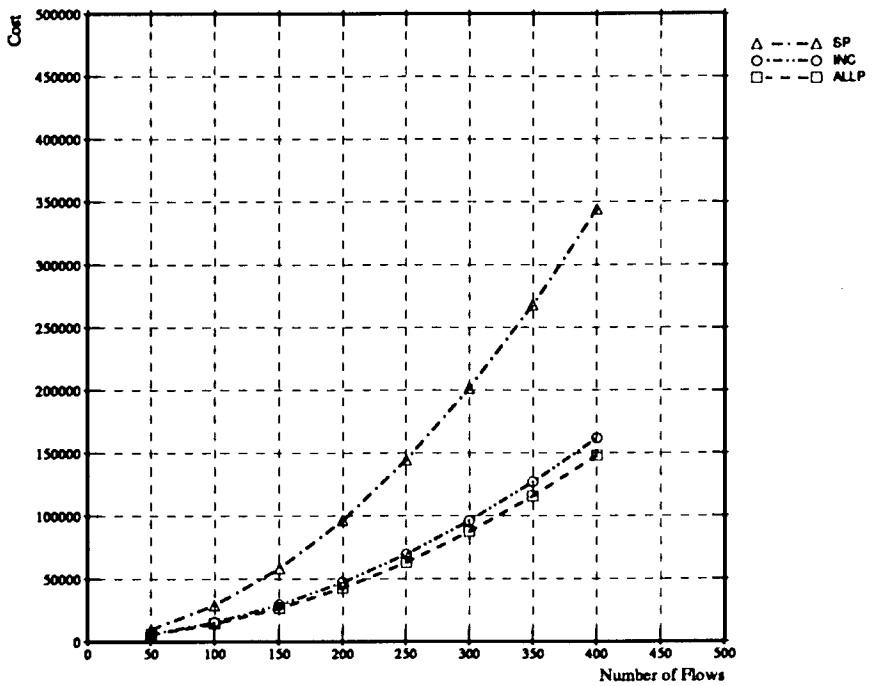

Figure 4.7: Comparison using the cost function: E-5 mesh, non-uniform distribution.

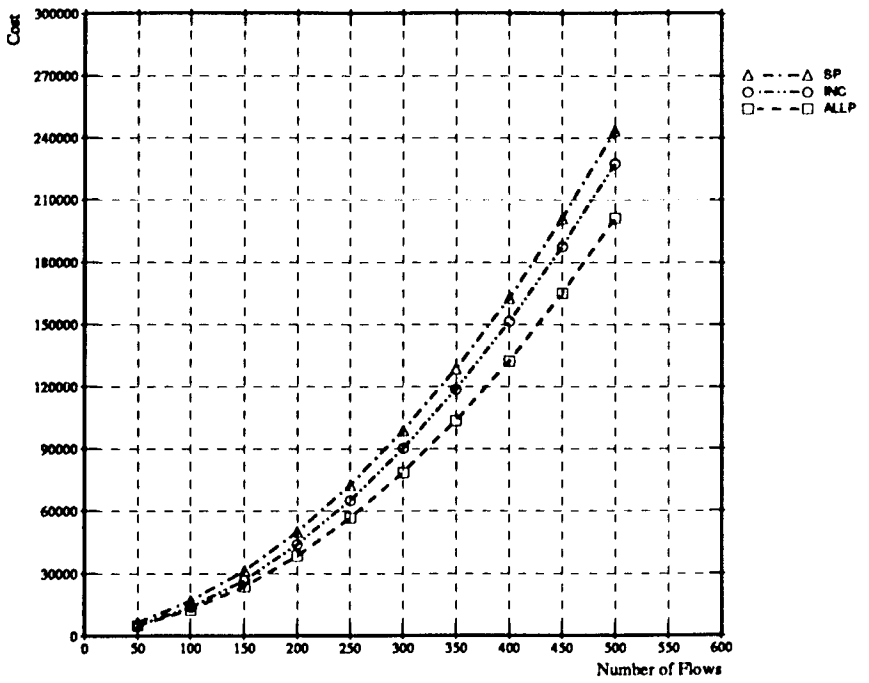

Figure 4.8: Cost comparison: E-4 mesh, uniform distribution.

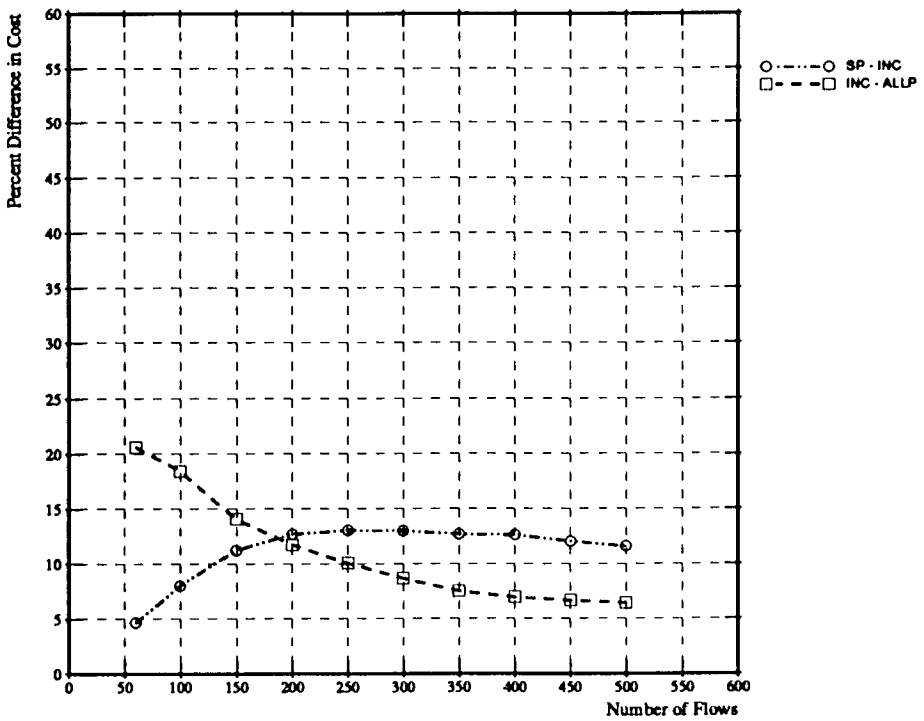

Figure 4.9: Performance improvement: E-4 mesh, uniform distribution.

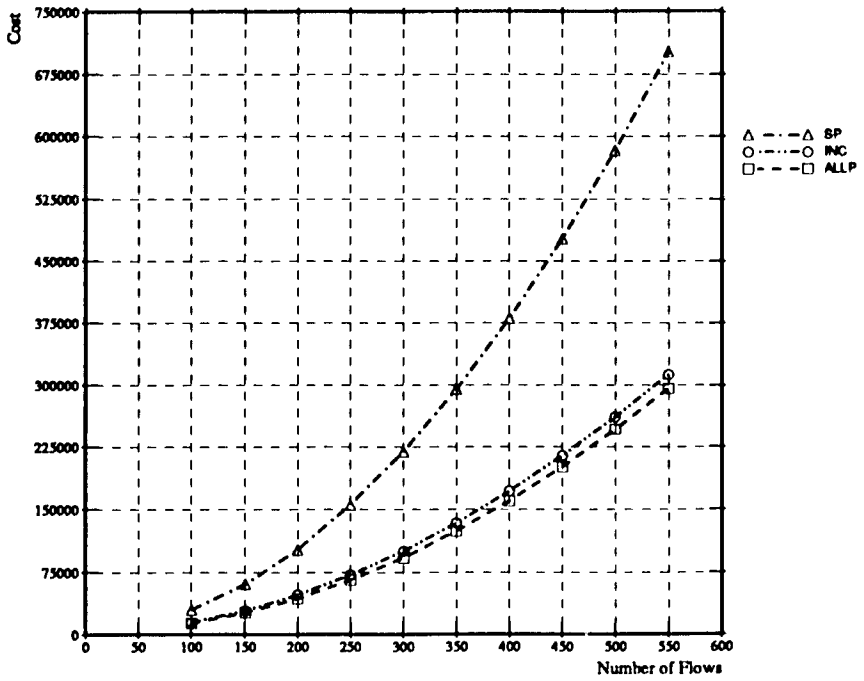

Figure 4.10: Cost comparison: E-4 mesh, non-uniform distribution.

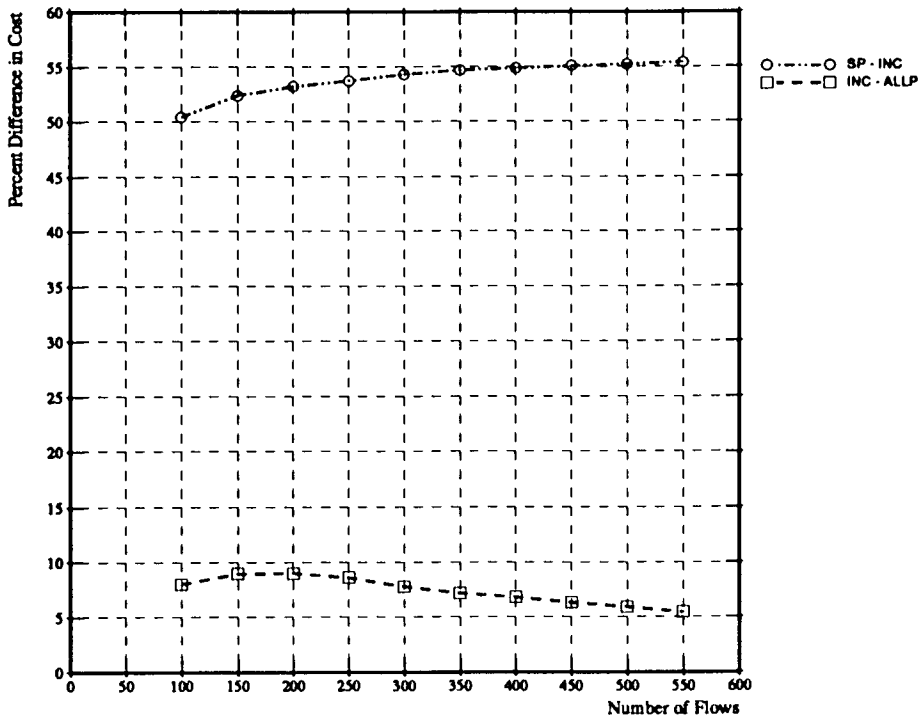

Figure 4.11: Performance improvement: E-4 mesh, non-uniform distribution.

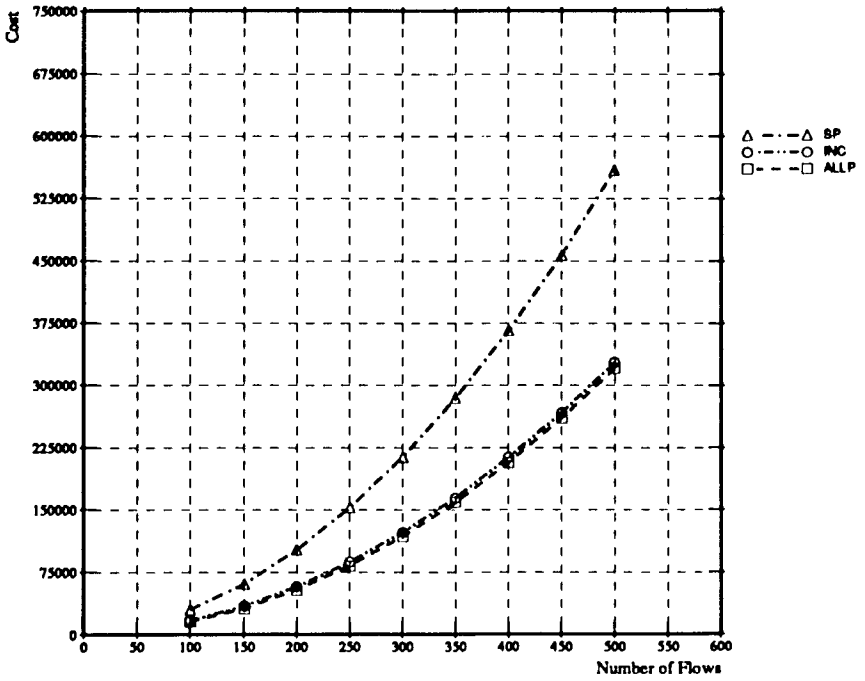

Figure 4.12: Cost comparison: Q-5 hypercube, uniform distribution.

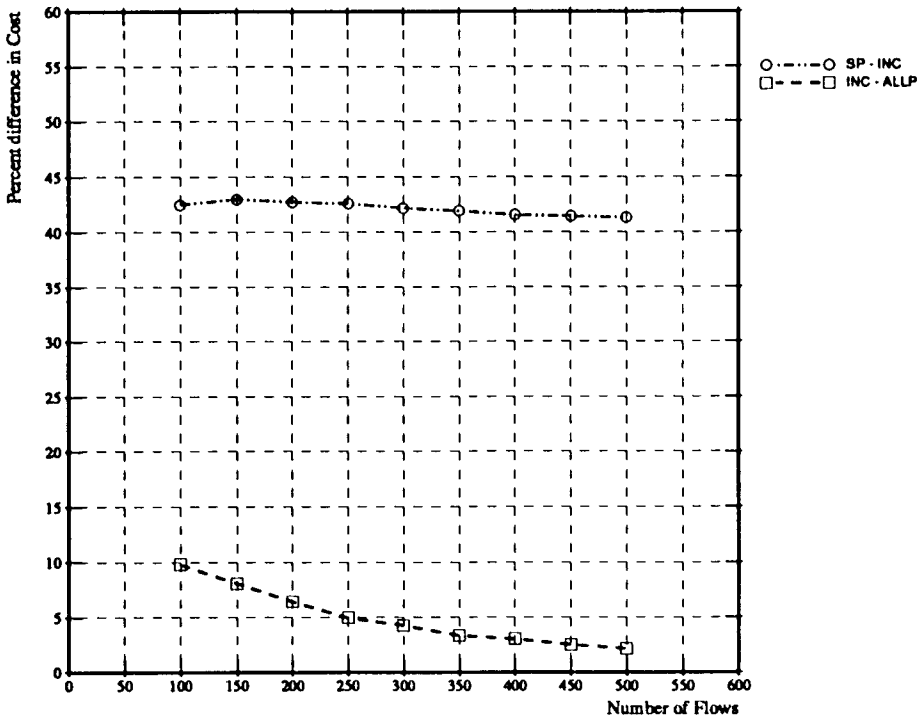

Figure 4.13: Performance improvement: Q-5 hypercube, uniform distribution.

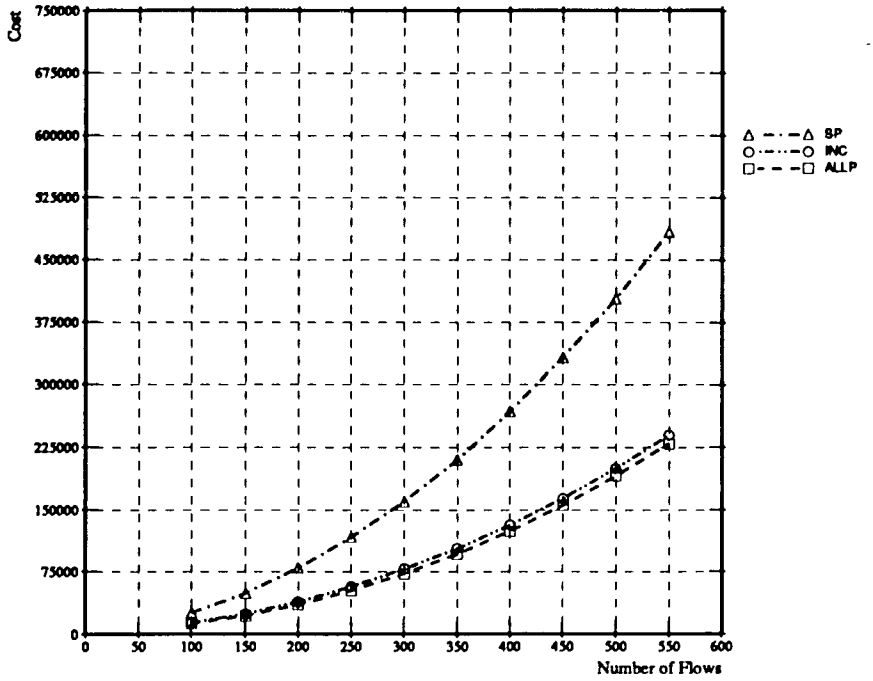

Figure 4.14: Cost comparison: Q-6 hypercube, uniform distribution.

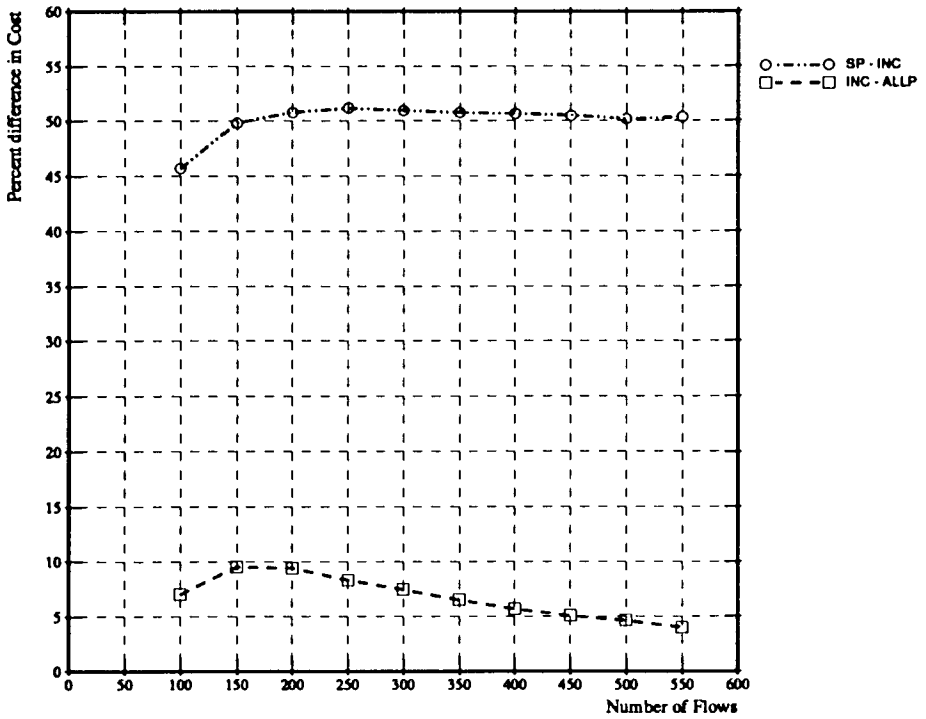

Figure 4.15: Performance improvement: Q-6 hypercube, uniform distribution.

CHAPTER 5

RELIABLE BROADCASTING

5.1 Introduction

This chapter addresses the problem of broadcasting in mesh-connected multicomputer systems, like HARTS, which use virtual cut-through switching. Although this operation is very simple for *broadcast* networks like the Ethernet and the Token Ring, where a message transmitted can be 'seen' by every other node in the network, it is more involved for a point–to–point interconnection network. For this type of network, a simple non-redundant broadcast algorithm, which delivers a single copy of a message to every node, essentially constructs a spanning tree for the network graph rooted at the source node. It is desirable to minimize the height, and hence the number of store–and–forward communication steps, for the spanning tree. We present a simple primitive to support broadcasting efficiently in this type of network. This primitive is based on the virtual cut-through switching scheme and significantly reduces the number of required store–and–forward communication steps. We also present an implementation technique for this primitive for the HARTS routing controller.

Based on this primitive, we then develop broadcasting algorithms for meshes which are resilient to node/link faults. Motivation for this work is provided by its applicability in implementing algorithms for problems like clock synchronization and distributed agreement in the presence of faults [LMS85, LSP82]. In these problems, it is necessary to ensure that a non-faulty node can correctly deliver its private value to all other non-faulty nodes in the system. This problem is difficult because (intermediate) faulty nodes can discard, corrupt, and possibly alter the information passing through them. Although on-line distributed diagnosis schemes are available for identifying faults, these schemes do not give 100% fault coverage unless the testing for the diagnosis goes on for a very long time. Therefore, it is highly desirable that these broadcast algorithms should work even when the identity of all the faulty processors is not known. This is accomplished by delivering multiple copies of the

message through disjoint paths to every node in the system. The receiving nodes can then identify the original message from the multiple copies using a scheme which is appropriate for the fault model, like majority voting.

Although the broadcast primitive can be used to develop similar algorithms for rectangular meshes, we only present algorithms for the wrapped hexagonal mesh topology, since these are more complex than those for a rectangular mesh.

To the best of our knowledge, this is the first reported work dealing with reliable broadcasting in point–to–point interconnection networks with virtual cut-through switching. In other related work, Chou and Gopal have recently presented some algorithms for linear broadcast routing [CG89]. The linear broadcast technique is similar in principle to the broadcasting primitive presented here. The authors, however, concentrate on the problem of finding optimal simple broadcast algorithms for general network topologies, and they have shown that the general problem is NP-complete. A multiple-copy reliable broadcast algorithm for the hypercube topology is presented in [RS88]. Algorithms for total-exchange and optimal broadcasting, again in hypercube multi-computers, can be found in [Fra89, JH89]. In [CSK90] we presented a point–to–point broadcast algorithm for the hexagonal mesh, which required $n+2$ communication steps in a mesh of size n. That algorithm, which is based on traditional store–and–forward switching, does not consider possible hardware support for virtual cut-through switching, and it does not handle multiple-copy broadcasts. Protocols for reliable broadcasting, mainly for broadcast networks, can also be found in the literature [CM84, BJ87]. These protocols try to provide a consistent delivery ordering among broadcast messages, but they do not consider malicious failures.

This chapter is organized as follows. Section 5.2 describes the proposed broadcast primitive and its implementation for HARTS. In Section 5.3 we develop an algorithm for simple broadcasting based on this primitive. Broadcast algorithms, which deliver multiple copies of the message through node-disjoint paths to each node in the hexagonal mesh, are presented in Section 5.4. An analysis of these algorithms and their comparison with other broadcast algorithms is presented in Section 5.5. The chapter concludes with Section 5.6.

5.2 The Broadcast Primitive

When we consider multi-computer systems with virtual cut-through switching, packet routing is typically handled by a front-end controller at each node. The normal operation of the controller is to compare the packet destination with the node address and if they match, the packet is delivered to the processor. Otherwise, it is forwarded to the

next node in the route. In many such systems dynamic routing is employed, in which case the controller also has to choose the next node on the route. For example, in HARTS, the possible routes of a message to the destination are described by three routing tags (which take positive and negative values) corresponding to the distances to be traversed in the six directions in the hexagonal mesh. The routing controller examines the tags in turn to check whether there are any non-zero values, and if so, whether the corresponding outgoing link is available. As the message is being routed toward the destination, its routing tags are also updated to reflect the new distance to the destination.

One of the principal advantages of this scheme is that the node processor does not have to examine and process all the packets going through the node. However, this advantage would be lost when broadcast messages are to be delivered using a simple store–and–forward broadcasting scheme. In addition to the larger delays caused by buffering, these messages could result in a substantial load on the processors. To facilitate efficient broadcasting, it is therefore necessary to support the operation at the link level. We propose to use the RELAY primitive, shown below in the form of a procedure, to accomplish this. This procedure shows the actions to be taken by the link controller when a packet arrives. It is assumed that the packet header contains the information required for handling broadcast messages like *type*, *distance*, *step*, and *tag*. The *type* field distinguishes a BROADCAST packet from an ordinary packet, while the *distance* gives the number of nodes to be traversed in a particular direction. The *step* and *tag* fields are used by the broadcast algorithms described later in this chapter.

In the RELAY procedure, *deliver* corresponds to the delivery of the packet by the link controller to the processor. The procedure also shows that the link controller is responsible for updating the distance field in the packet header before delivering or relaying the packet. The packet is relayed to the next node in the same direction in which it arrived, i.e., on the link opposite to the input one, using *send_on_link*.

```
procedure RELAY

begin
    receive_from_link(packet, from_direction)
    if (packet.type = BROADCAST)
        packet.distance := packet.distance − 1
        deliver(packet)
        if (packet.distance ≠ 0)
            send_on_link(direction = from_direction, packet)
        end
    else
        normal packet handling
    end
end
```

There are several reasons for choosing this primitive. First, it blends in easily with the existing dynamic routing algorithms. Second, the *deliver* and the *send_on_link* steps can be accomplished concurrently using a "tee" operation. Third, the operation is simple enough to be implemented at little additional cost in the link controller. Furthermore, we will show that this primitive can be used very effectively to develop broadcast algorithms for mesh connected multi-computers.

The implementation of the "tee" operation can be described in more detail in the context of the HARTS routing controller [DRS89]. The controller contains six receivers and six transmitters, corresponding to the incoming and outgoing links, connected to a single bus. This bus, called the *time-sliced bus*, also has interfaces to the packet buffer management unit in the node to accept and deliver packets. The bus is time-slotted and each receiver is thus guaranteed an access slot, which it uses to place the data that it receives on the bus. Most of the intelligence in the routing controller resides in the receivers. When a packet is received, the receiver examines the routing tags in the packet header to check whether the packet has reached its destination. If not, it checks the directions in which a packet can be forwarded and tries to reserve a transmitter in one of these directions. Note that when shortest path routing [CSK90] is used, the routing tags are such that at most two of the three routing tags are non-zero. In this case, the packet can be forwarded in at most two of the six directions. If the reservation succeeds, the transmitter accepts any data that is placed on the time-sliced bus by the receiver and transmits it. If the reservation attempts do not succeed, the receiver asserts a control line to request the buffer management unit to store the packet for later transmission.

To implement the RELAY primitive, the receiver operation can be modified to recognize packets of type BROADCAST. For this packet type, in addition to attempting a reservation for the transmitter in the same direction, it also asserts the control line to store

the packet. Therefore, when the receiver places packet data on the bus, it can be forwarded to the next node (*send_on_link*) and dropped to the node (*deliver*) simultaneously. If the reservation does not succeed, the packet is dropped to the buffer management unit as usual. In practice, only one packet is delivered to the buffer management unit even if the packet cannot cut through to the next node. The packet header is marked appropriately to inform the network processor about the status of the forward transmission. The HARTS routing controller is microprogrammable and these modifications have been implemented by changing the micro-programs, without any change to the controller hardware.

5.3 Simple Broadcasting

The algorithm for simple broadcasting is shown below in the form of procedures BCAST_INIT and SBCAST_RELAY. An example of its operation is given in Figure 5.1 for a hexagonal mesh of size 4 (denoted by E-4). In this algorithm, and in the algorithms described later in this section, the size of the hexagonal mesh is n and the directions referred to are labeled in a counter-clockwise sense, as illustrated in Figure 5.2(a). With reference to the definition of the C-wrapped hexagonal mesh, direction 0 corresponds to the link from a node s to the node $[s+1]_{3n^2-3n+1}$. The term *principal axis* is used frequently in the explanation of the algorithms. This refers to an imaginary line connecting the center of the hexagon to one of the six corners. Since the C-wrapped hexagonal mesh is a homogeneous structure, any node can be considered to be at the center of the mesh and the algorithms can be described by placing the broadcasting node at the center. It is also useful to define directions relative to the direction in which the packet arrived into a node, as in Figure 5.2(b). Hence, *left* corresponds to the absolute direction (in + 1) mod 6, *right* corresponds to direction (in − 1) mod 6, and so on.

The procedure BCAST_INIT is executed by the node which initiates the broadcast, and is common to all broadcasting algorithms. This node plays no further part in the broadcast process. In BCAST_INIT, the distance is set to $n-1$ because this is the diameter of a hexagonal mesh of size n. The *send_packet* function, which is also used in other algorithms, is a non-blocking *send* and only initiates the transmission of the packet. Actual packet transmission can proceed in parallel on the six outgoing links after the initiation. The other procedure, SBCAST_RELAY, is not specific to a particular node and describes the overall operation for the system. It is activated whenever a broadcast packet is received at a node. The actions taken by a node are driven by the information that it receives in the broadcast packet. It is noted that the step number, *step*, and the algorithm type are

a part of the state information that is contained in the broadcast packet. Moreover, the information about the direction from which a particular packet was received (denoted as *from_direction*), is available to the receiving node. This is indicated by the *receive* operation in SBCAST_RELAY. Based on this information, the processor at an intermediate node can determine the next step in the broadcast as per the SBCAST_RELAY algorithm. The algorithm terminates after step 2 because packets with a step value of 2 do not branch any further. Note that *send_packet* and *receive* are processor level operations, distinct from the link level operations shown in the RELAY primitive.

procedure BCAST_INIT

```
begin
    packet.type := BROADCAST
    packet.step := 1

    send_packet(packet, direction=0, distance=n − 1)
    send_packet(packet, direction=1, distance=n − 1)
    send_packet(packet, direction=2, distance=n − 1)
    send_packet(packet, direction=3, distance=n − 1)
    send_packet(packet, direction=4, distance=n − 1)
    send_packet(packet, direction=5, distance=n − 1)
end
```

procedure SBCAST_RELAY

```
begin
    receive(packet, from_direction)
    if (packet.step = 1)
        packet.step := 2
        if (packet.distance ≠ 0)
            direction := (from_direction + 1) mod 6
            send_packet(packet, direction, packet.distance)
        end
    end
end
```

The correctness of this algorithm can be explained based on Figure 5.1, which shows the paths taken by a broadcast packet. The broadcast packet is delivered to all nodes on the six principal axes by the BCAST_INIT operation. The "distance" field in the broadcast packet header is decremented, as shown in the RELAY primitive, at each intermediate node and at the receiving node. Hence, a node on the principal axis which is m hops away from the source node sees a value of $(n - 1 - m)$ in the *packet.distance* field. In SBCAST_RELAY, this value is used in the forwarding, so the forwarded packet travels a total distance of $m + (n - 1 - m) = n - 1$ from the source node. Since the nodes on the periphery of the hexagonal mesh are $n - 1$ hops from the center, the forwarded packet will reach the peripheral node.

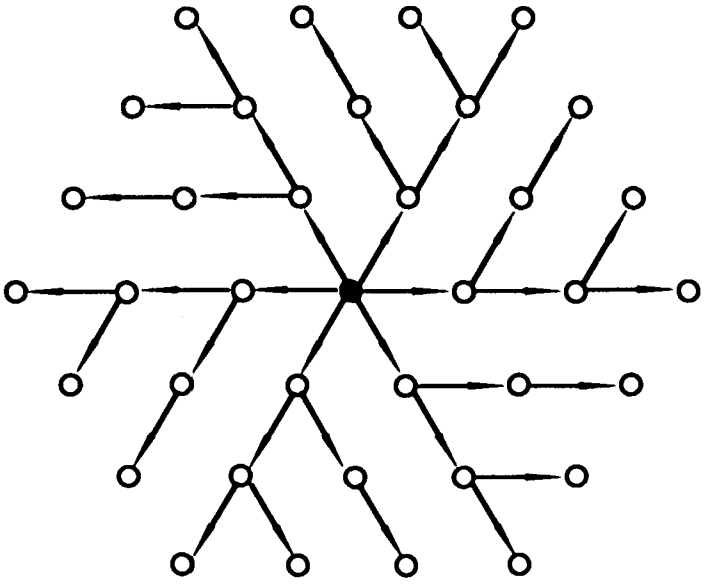

Figure 5.1: Simple broadcast for an E-4 mesh (SBCAST).

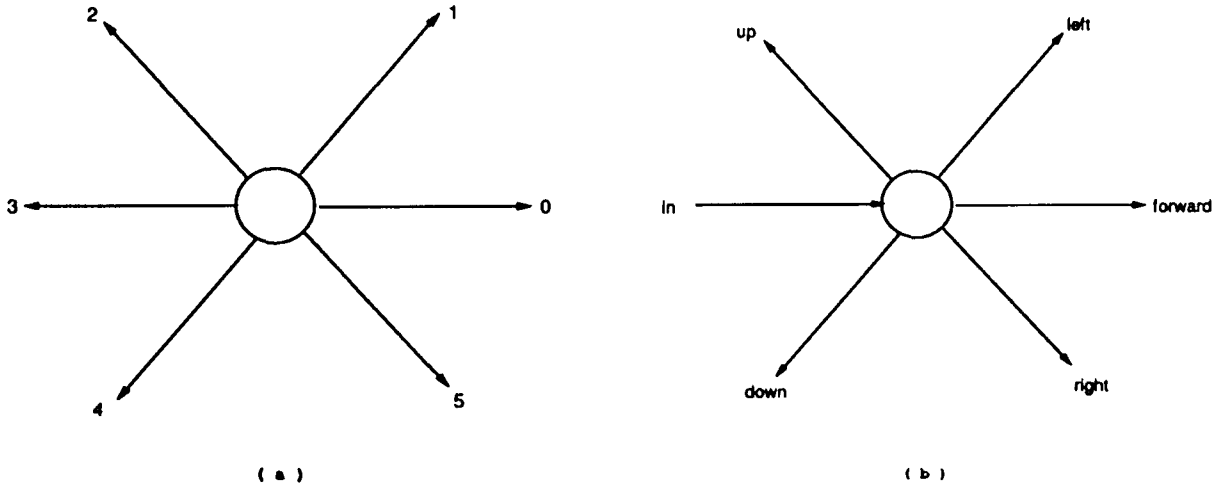

Figure 5.2: Direction labeling.

5.4 Multiple Copy Broadcasts

While simple broadcasting is sufficient for many applications, it is susceptible to message loss, possibly due to data corruption and/or link and node failures. There are several applications which require more resilient broadcast mechanisms, like the clock synchronization algorithm described in [RKS90]. For this type of application, we have developed a family of efficient and elegant algorithms, called *k-reliable* broadcasts, to deliver multiple copies of a message to each node using node-disjoint paths. The algorithms can also be used to guard against message loss in applications which require reliable message delivery, in place of the conventional acknowledgment–retry mechanism.

In the clock synchronization algorithm, for example, to tolerate m arbitrary (Byzantine) faults, it is necessary for a node to transmit $2m + 1$ copies of its local clock to every other node in the system through disjoint paths. From the values received, a node can determine the value that was sent by the originator using the technique described in [YM88]. Therefore, using a 5-reliable broadcast, it is possible to achieve clock synchronization in a hexagonal mesh in the presence of up to two Byzantine faults. In this application, and in other applications of reliable broadcasting, there are two aspects to a reliable broadcast: the *delivery* mechanism and the *reception* mechanism[1]. The delivery mechanism consists of algorithms that deliver multiple copies of a message to all other nodes, through disjoint paths. It is noted that in the presence of faults, some of these copies may be corrupted or lost. The reception mechanism involves algorithms which interpret and assimilate information from the different copies received at a node. These are strongly dependent on the fault model used. A discussion on different reception mechanisms can be found in [RS88]. These mechanisms are not dependent on the hypercube topology, so they can also be used for the hexagonal mesh.

This section presents the message delivery algorithms, starting from the two-copy algorithm and progressing toward the six-copy version. The delivery algorithms have a common broadcast initiation procedure, BCAST_INIT, which was described in Section 5.3. Figure 5.3 shows the packets generated in one direction for the different broadcast algorithms, where the source node is placed at the lower left corner of the hexagonal mesh.

---

[1]The terminology used here is taken from [RS88].

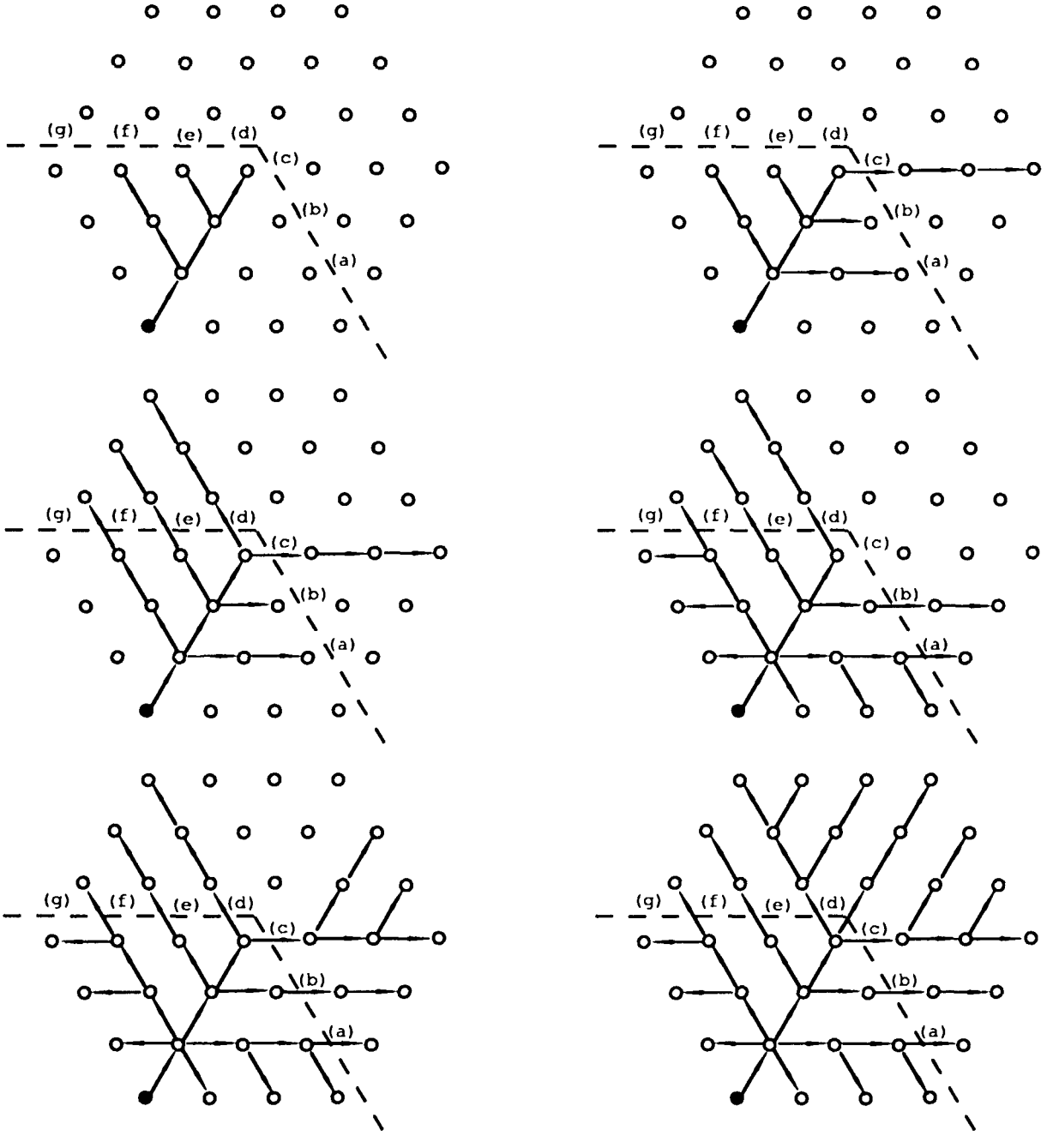

Figure 5.3: Packets generated in one direction for the six broadcast algorithms.

5.4.1 2-Reliable Broadcast (2-BCAST)

The algorithm 2-BCAST shown below delivers two copies of the message to each node. Its operation is illustrated in Figure 5.4 for an E-4 mesh, and for clarity, only the actions of nodes on two of the principal axes are shown. Also, the links that wrap around are shaded and labeled. This algorithm is similar to SBCAST_RELAY, except that the message is forwarded in two directions. Using the explanation presented earlier for the simple broadcast algorithm, it is observed that nodes which are not on the principal axes get two copies of the message, as shown in the figure. Also, nodes on the extremities of the principal axes (*packet.distance* = 0) use the wrap links to send the message to nodes on another axis. For example, in the figure, the wrap links (a) and (b) are used to deliver messages to two other principal axes. This ensures that nodes on the principal axes also get two copies of the message, and through disjoint paths.

procedure 2-BCAST

```
begin
    receive(packet, from_direction)
    if (packet.step = 1)
        packet.step := 2
        if (packet.distance ≠ 0)
            send_packet(packet, direction=(from_direction + 1) mod 6, packet.distance)
            send_packet(packet, direction=(from_direction − 1) mod 6, packet.distance)
        else
            send_packet(packet, direction=(from_direction − 1) mod 6, distance=n − 1)
        end
    end
end
```

5.4.2 3-Reliable Broadcast (3-BCAST)

The 3-reliable broadcast algorithm can be explained through a transformation of the 2-BCAST algorithm. In going from 2-BCAST to 3-BCAST, it is necessary to deliver one more copy of the packet to each node. This is accomplished by arranging for the delivery of the third copy using the wrap links from nodes that are diametrically opposite with respect to the broadcasting node. There are two modifications required to the 2-BCAST algorithm, intermediate nodes on the principal axes now transmit the packet *left* for $n-1$ hops to reach nodes in the opposite sextant. Also, the extreme nodes on the axes transmit the packet $n-1$ hops to the *left*. The results of these modifications can be seen in Figure 5.5, where the labels indicate nodes connected by a wrap link.

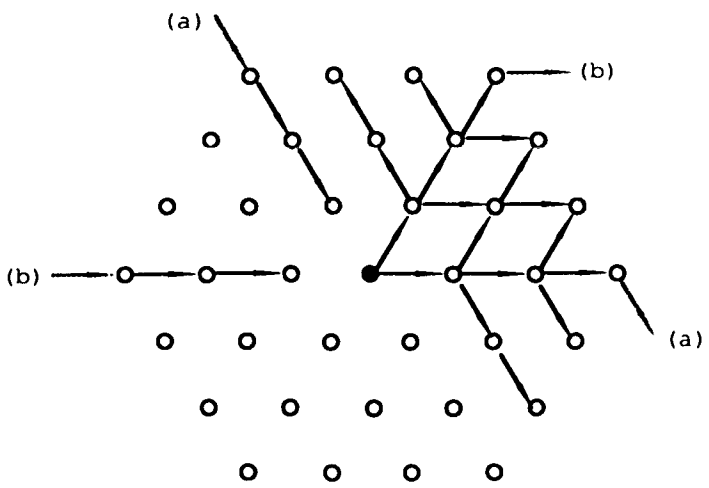

Figure 5.4: 2-BCAST for an E-4 mesh.

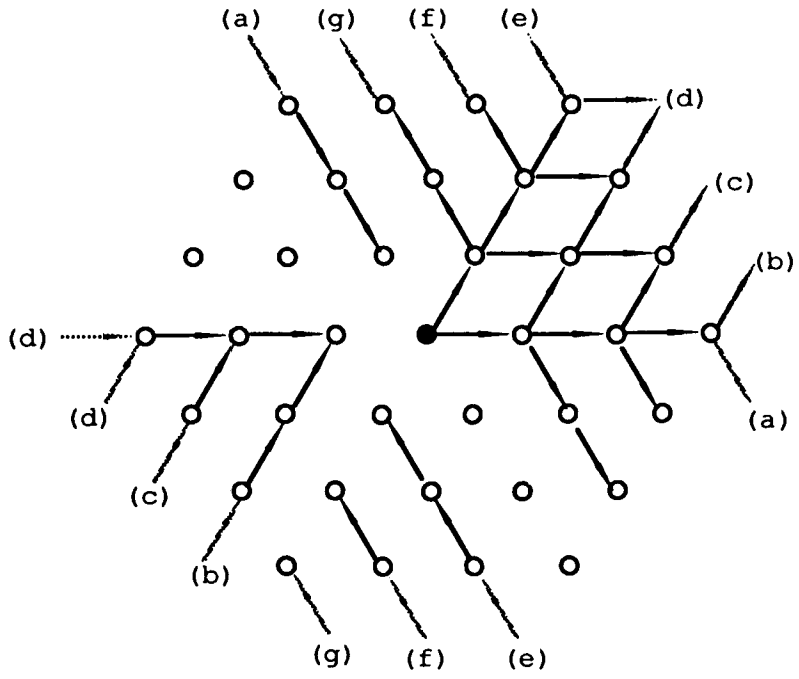

Figure 5.5: 3-BCAST for an E-4 mesh.

procedure 3-BCAST

```
0. begin
1.    receive(packet, from_direction)
2.    if (packet.step = 1)
3.          packet.step := 2
4.          if (packet.distance ≠ 0)
5.                send_packet(packet, direction=(from_direction + 1) mod 6, distance=n − 1)
6.                send_packet(packet, direction=(from_direction − 1) mod 6, packet.distance)
7.          else
8.                send_packet(packet, direction=(from_direction + 1) mod 6, distance=n − 1)
9.                send_packet(packet, direction=(from_direction − 1) mod 6, distance=n − 1)
10.         end
11.   end
12. end
```

5.4.3 6-Reliable Broadcast (6-BCAST)

We choose to describe 6-BCAST before 4-BCAST and 5-BCAST because the latter two can be treated as restricted forms of 6-BCAST. This algorithm, which is presented in procedure 6-BCAST, creates the broadcast tree shown in Figure 5.6. This figure shows the packets generated by 6-BCAST from a single principal axis. As compared to 3-BCAST, this algorithm is more complicated because it is necessary to use an additional forwarding step at some of the nodes. We employ a *tag* field in the broadcast packet header to ensure that only the relevant nodes will execute the additional step. This tag field takes different values, and it is interpreted by the receiving node to forward the packet in the appropriate direction (lines 26–38 of 6-BCAST). Nodes on the extremities of the principal axes use tags 'A' and 'B' to forward the packet to two sextants. This is shown in the first part of Figure 5.6, where the labels mark nodes that are connected by wrap links. The immediate neighbors of the broadcast source node, with *packet.distance* of $n - 2$, use tags 'C' and 'D' to reach nodes on the adjacent principal axes. In the second part of Figure 5.6, the graph is redrawn by re-positioning the nodes reached by wrap links. The initiating node is now at the bottom left corner of the hexagon. The figure demonstrates that the broadcast tree generated is quite regular, which is not immediately apparent from procedure 6-BCAST.

It can be seen that the packets reach all the nodes in the hexagonal mesh (except the initiator node). Similarly, the nodes also receive packets originating from each of the other five principal axes. Hence, each node receives six copies of the packet. However, it is not obvious from the figure that these six copies would be received through *node disjoint* paths. We will show that this is indeed the case later in the section. Given that the node degree of all nodes of the hexagonal mesh is six, this is the maximum number of disjoint paths possible. *This algorithm shows that the hexagonal mesh is 6 connected in terms of*

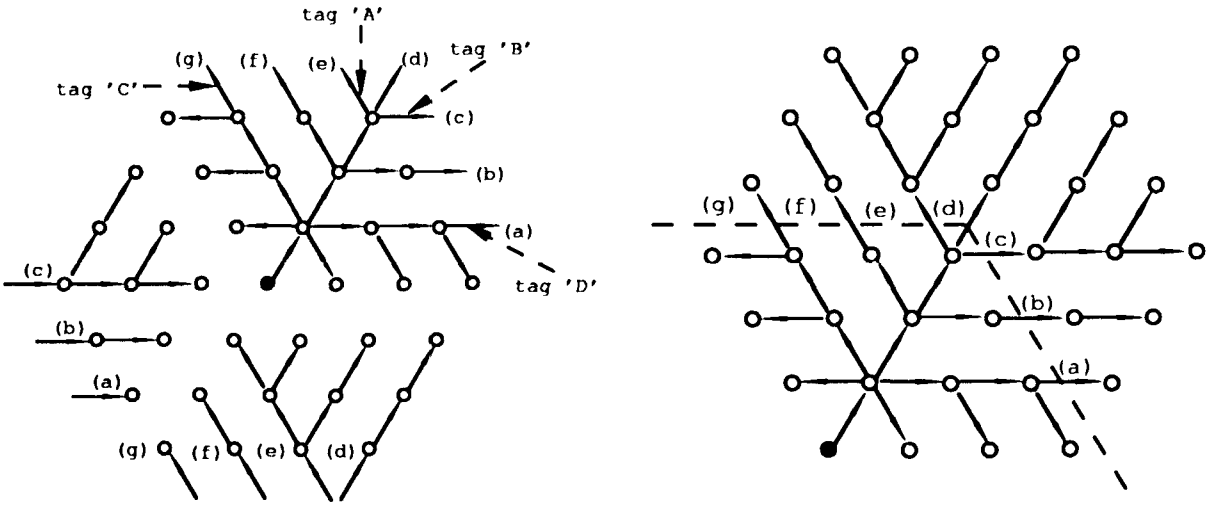

Figure 5.6: Packets generated in a 6-BCAST (from one direction).

node connectivity.

5.4.4 5-BCAST and 4-BCAST

Both 5-BCAST and 4-BCAST can be realized as restricted forms of 6-BCAST. In 5-BCAST, we eliminate the additional forwarding step for packets that were tagged 'A' in 6-BCAST. We can accomplish this by setting *packet.tag* to NONE on line 5 in procedure 6-BCAST, and by excluding the *send* operation on line 10. The resulting broadcast tree, in one direction, is shown in Figure 5.7. To get 4-BCAST from 5-BCAST, we further eliminate the *send* operation on line 7 of 6-BCAST. The broadcast tree for 4-BCAST is shown in Figure 5.8.

5.4.5 Correctness of the Algorithms

The correctness of the simple broadcast algorithm can be shown using the figure of the complete broadcast tree (Figure 5.1). This technique can also be used for 2-BCAST, to show that each node receives two copies and the paths used are node disjoint. For the more complicated algorithms like 6-BCAST, we can show that all nodes receive the required number of copies using the broadcast tree. To show that the paths are node disjoint, we consider the case of 6-BCAST in some detail and examine the paths generated from the source node to a particular destination node. There are two main cases to be considered: (1) the destination node is on one of the principal axes, (2) the destination node is between two principal axes. Figure 5.9 shows the paths in these two cases.

procedure 6-BCAST

```
0. begin
1.    receive(packet, from_direction)
2.    if (packet.step = 1)
3.          packet.step := 2
4.          if (packet.distance = 0)
5.                packet.tag := 'A'              /* tag = NONE for 4-BCAST and 5-BCAST */
6.                send_packet(packet, direction=(from_direction + 1) mod 6, distance=n − 1)

7.                packet.tag := 'B'              /* this send is excluded for 4-BCAST */
8.                send_packet(packet, direction=(from_direction − 1) mod 6, distance=n − 1)

9.                packet.tag := NONE             /* this send is excluded for 4-BCAST and 5-BCAST */
10.               send_packet(packet, direction=from_direction, distance=n − 1)
11.         else if (packet.distance = n − 2)
12.               packet.tag := 'C'
13.               send_packet(packet, direction=(from_direction + 1) mod 6, distance=n − 1)

14.               packet.tag := 'D'
15.               send_packet(packet, direction=(from_direction − 1) mod 6, distance=n − 1)

16.               packet.tag := NONE
17.               send_packet(packet, direction=(from_direction + 2) mod 6, distance=1)
18.               send_packet(packet, direction=(from_direction − 2) mod 6, distance=1)
19.         else
20.               packet.tag := NONE
21.               send_packet(packet, direction=(from_direction + 1) mod 6, distance=n − 1)
22.               send_packet(packet, direction=(from_direction − 1) mod 6, distance=n − 1)
23.         end
24.   else if (packet.step = 2) AND (packet.distance ≠ 0)
25.         packet.step = 3
26.         case (packet.tag) of
27.               'A':
28.                     send_packet(packet, direction=(from_direction − 1) mod 6, packet.distance)
29.               'B':
30.                     send_packet(packet, direction=(from_direction + 1) mod 6, packet.distance)
31.               'C':
32.                     send_packet(packet, direction=(from_direction + 1) mod 6, 1)
33.               'D':
34.                     send_packet(packet, direction=(from_direction − 1) mod 6, 1)
35.               NONE:
36.         end
37.   end
38. end
```

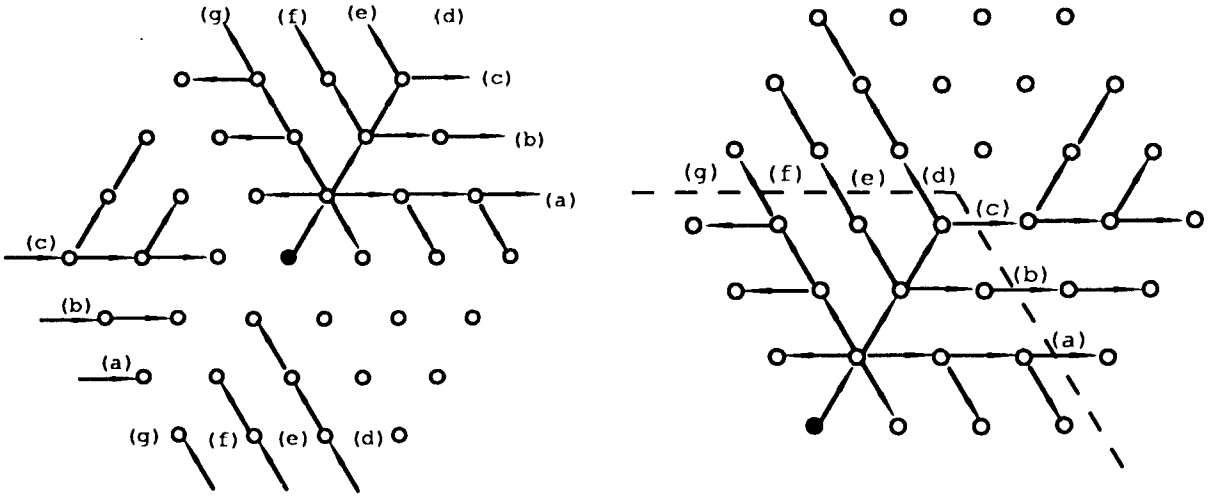

Figure 5.7: Packets generated in a 5-BCAST (from one direction).

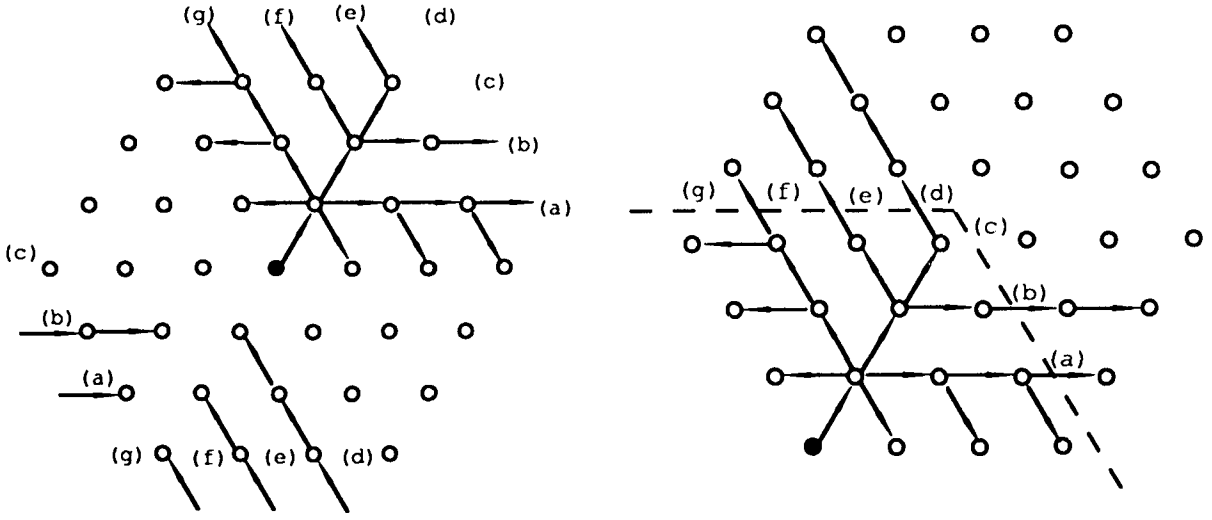

Figure 5.8: Packets generated in a 4-BCAST (from one direction).

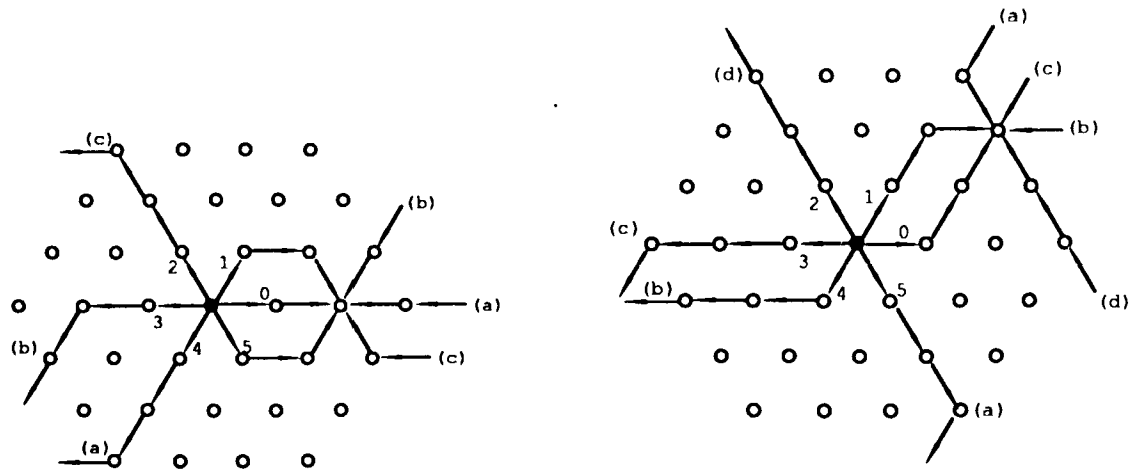

Figure 5.9: Disjoint paths in a 6-BCAST

Case 1: Path 0 is created by the BCAST_INIT operation. Paths 1 and 5 are created using tag 'D' and tag 'C' packets (lines 13 and 15), with subsequent forwarding. Path 2 is created using a tag 'A' forwarding on lines 6 and 28, whereas path 3 is created on line 21. Path 4 is created by the send operation on line 6.

Case 2: Paths 0 and 1 are created from the axes which bound the sextant containing the destination node. They are created essentially by the operations on lines 21 and 22. Versions of these operations for nodes adjacent to the sender (distance = $n-2$) and at the end of the principal axes (distance = 0) can be found on lines 13, 15 and 6, 8 respectively. Paths 3 and 4 are similar, and they are created from the opposite axes. The send operation on line 10 creates path 2. Path 5 is created using tagged forwarding on line 6.

We can see that the 6 paths generated in each case are node disjoint. 4-BCAST and 5-BCAST are restricted forms of 6-BCAST, so it follows that the paths generated by these algorithms are also node disjoint. In 5-BCAST, one sextant of the hexagon is left uncovered in each direction. From the structure of the broadcast tree, we can see that the a different sextant is left uncovered in each direction. Hence, the number of copies received by each node is five. Similarly, in 4-BCAST, the number of copies received is four for the 4-BCAST, and the paths are disjoint.

We have also used an enumeration technique to independently verify these algorithms. We developed a program which implements the broadcast algorithms and generates broadcast trees for a fixed source node. From the broadcast tree of the k-BCAST, this program traces the paths from the source node to each node in the system. It then verifies that each node is visited by k paths and that the paths are node disjoint. We used this

program to verify that the algorithms performed correctly, for mesh sizes[2] from 3 to 15.

5.5 Algorithm Analysis

The *latency* of a broadcast can be defined as the elapsed time between the initiation of the broadcast and the delivery time of the last packet in the broadcast. This latency can vary depending upon the system load and the number of cut-through routes. To analyze broadcast algorithms, one metric is the latency for the algorithm in the best case, that is, when the network is otherwise idle. This latency can be computed based on a model that is commonly used for point-to-point communications. The time required to transmit a packet of length M can be modeled as $S + rM$, where S is the packet set-up time and r is the transmission rate on the link. When a packet *cuts through* a node, the delay experienced is essentially the time taken to receive and examine the packet header, a small constant d. Hence, if a packet cuts through i nodes, the time elapsed between the start of transmission and the end of reception is $S + rM + id$.

Consider the simple broadcast algorithm presented in Section 5.3. Assuming that the network is otherwise idle, and the node can concurrently transmit messages on multiple links (as in the HARTS routing controller), the packet can be delivered to all nodes on the principal axes in a single transmission. The second step, which completes the operation, can also be accomplished by a single packet transmission. Hence, in the best case, the broadcast operation can be completed using two packet transmissions. The longest path in this broadcast is $n - 1$ hops, which is the diameter of a hexagonal mesh of size n, and the packet is buffered and relayed only once on this path. Hence, the number of nodes that are cut through is $n - 3$ and the best-case maximum message latency for the broadcast is $2S + 2rM + (n-3)d$. The average-case maximum message latency is $(k+2)(S+rM)+(n-3-k)d$, where $k = (n-3)\rho$. This algorithm compares very favorably with the point-to-point broadcast algorithm presented in [CSK90], since that algorithm has latency $(n+2)(S+rM)$.

The latency for the other broadcast algorithms can be determined in a similar fashion, using the number of message transmissions, and the maximum path length. Note that the *send_packet* operations in all the algorithms can be performed in parallel since they do not have any common links. The paths traced by these algorithms do not result in any contention for the links because each link has to carry at most one packet. Moreover, as mentioned earlier, it is assumed that a node can transmit packets on more than one

[2] A hexagonal mesh of size 2 is a complete graph of 7 nodes, which can be treated as a special case

Type	Best-Case
SBCAST	$2(S + rM) + (n - 3)d$
2-BCAST	$2(S + rM) + 2(n - 2)d$
3-BCAST	$2(S + rM) + 2(n - 2)d$
4-BCAST	$3(S + rM) + (n - 3)d$
5-BCAST	$3(S + rM) + (2n - 5)d$
6-BCAST	$3(S + rM) + (2n - 5)d$

Table 5.1: Latency for different broadcast algorithms.

outgoing link simultaneously. This is the case for the HARTS routing controller [DRS89]. The latencies of these algorithms are shown in Table 5.1. 2-BCAST and 3-BCAST have a minimum number of *two* message transmissions, whereas the other three algorithms require three transmissions because of the additional forwarding step.

A comparison based only on the best case latency is not satisfactory because the latency is very sensitive to the number of times that a message gets buffered. In the C-wrapped hexagonal mesh, the RELAY primitive can also be used to send a packet to all nodes using *send_packet* with distance set to $3n(n-1)$, since this traces out a Hamiltonian cycle along any one of the six directions emanating from the broadcasting node. The latency for packet delivery for this algorithm (called Algorithm A) would be $S+rM+(3n(n-1)-1)d$ in the best case. This shows that in the best case, for certain values of S, r, and d, Algorithm A can perform better than SBCAST.

However, the probability of a packet getting buffered increases with the length of the path, which results in a larger latency. For example, consider a packet that is to be delivered to a node which is m hops away. If the packet cuts through all the intermediate $m-1$ nodes, the latency is $S + rM + (m-1)d$. However, if the packet gets buffered at i of the $m-1$ nodes on the path, the latency experienced would be $(i+1)(S+rM)+(m-1-i)d$, where the average queueing delay experienced before a packet is serviced is included in the setup time S. The average number of times that a packet gets buffered can be determined using the queueing network model, like the one used in [KK79, IM86]. Assuming that the network is uniformly loaded and the utilization of each link is ρ, the probability that a message gets buffered waiting for a link is ρ. For brevity, we call this the probability of buffering for a link. Since the probability of buffering for a link is independent of the probability of buffering for any other link, the number of times that a message gets buffered follows a binomial distribution. Hence, if a message has to traverse m links, the average number of times that it will get buffered is $m\rho$.

ρ		SBCAST (no cut-through)	Algorithm A
$n = 3$	0.00	$2(S + rM)$	$(S + rM) + 17d$
	0.05	$2(S + rM)$	$1.85(S + rM) + 16.15d$
	0.10	$2(S + rM)$	$2.70(S + rM) + 15.30d$
	0.15	$2(S + rM)$	$3.55(S + rM) + 14.45d$
	0.20	$2(S + rM)$	$4.40(S + rM) + 13.60d$
$n = 4$	0.00	$3(S + rM)$	$(S + rM) + 35d$
	0.05	$3(S + rM)$	$2.75(S + rM) + 33.25d$
	0.10	$3(S + rM)$	$4.50(S + rM) + 31.50d$
	0.15	$3(S + rM)$	$6.25(S + rM) + 29.75d$
	0.20	$3(S + rM)$	$8.00(S + rM) + 28.00d$

Table 5.2: Comparison of simple broadcast algorithms.

Based on this model, the average-case broadcast message latency for Algorithm A is $(\ell+1)(S+rM)+(3n^2-3n-1-\ell)d$, where $\ell = (3n(n-1)-1)\rho$. It is difficult to compute a similar expression for SBCAST because it has many parallel transmissions and the latency is determined by the delivery time of the last (slowest) message. However, it is possible to compute the latency for SBCAST assuming that the broadcast message gets buffered at each intermediate node. A comparison of these two algorithms for different values of ρ in hexagonal meshes of size 3 and 4 is given in Table 5.2. It is noted that the setup time S is an increasing function of the utilization ρ because it also includes the queueing delay. From this table, it is clear that SBCAST outperforms Algorithm A as the utilization ρ increases. Also, as n increases, SBCAST performs better even for very small values of ρ. The SBCAST algorithm can also be compared with an "ideal" simple broadcast algorithm, which would deliver messages to all nodes with a single message transmission. Since the diameter of the hexagonal mesh is $n-1$, the number of nodes that are cut through is $n-2$. Hence, this ideal algorithm would have a best-case latency of $S + rM + (n-2)d$. Thus, SBCAST is within a constant factor of the ideal algorithm.

Simulation Results

In order to study the performance of our broadcast algorithms, we have simulated these algorithms using a discrete-event simulator described in Section 4.6. This simulator has been modified and extended to implement the RELAY primitive in the routing hardware and the broadcast algorithms in the network processor.

One of the objectives of the simulation experiments was to get an estimate of the performance of the broadcast algorithms under different network load conditions. The

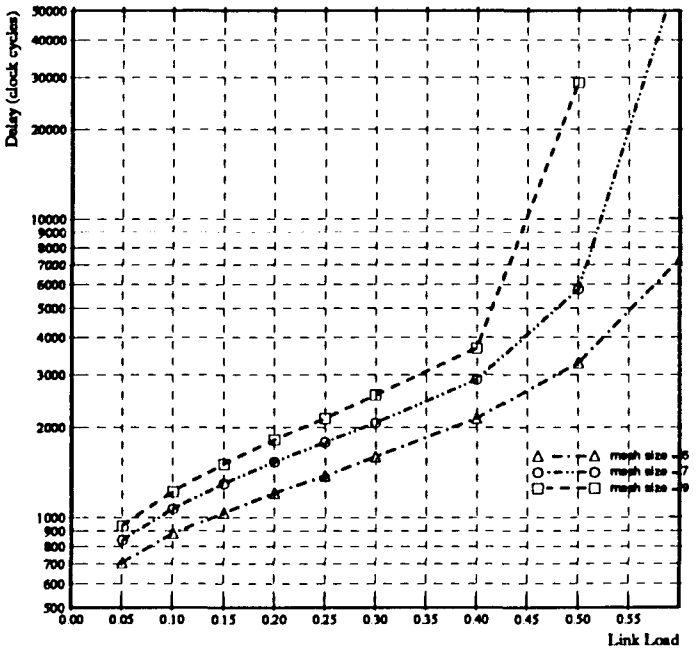

Figure 5.10: Performance of simple broadcast.

traffic generated for the network was uniform across all the nodes and consisted of two types of packets: regular and broadcast. At each source node, packets were generated using a Poisson arrival process and they were assigned type regular or broadcast with probability 0.999 and 0.001, respectively. For the base non-broadcast traffic, destination nodes were chosen such that the probability of communicating with a node was inversely proportional to the distance from the source. Both types of packets were assigned lengths of 64, 128, and 512 bytes, with probability 0.3, 0.5, and 0.2, respectively. The link load used for plotting is the ratio of the packet generation rate to the peak I/O rate of the routing hardware. Currently, the peak I/O rate that can be supported by the routing hardware is 4 MBytes per second.

Figure 5.10 shows the latency of message delivery for the SBCAST algorithm. The units for latency in this and other figures is hardware "clock cycles", in the current routing controller hardware this is 1.5 microseconds. The three curves shown in the figure are for meshes of size 5, 7, and 9. It can be seen that the latency increases with mesh size and with link load. For a particular link load the increase is close to linear when we move from one mesh size to another. The latency increases super-exponentially for link loads beyond 0.5, indicating that the network is close to saturation at that point. The reason for this "early" saturation is that link access overheads are not included in the computed peak I/O

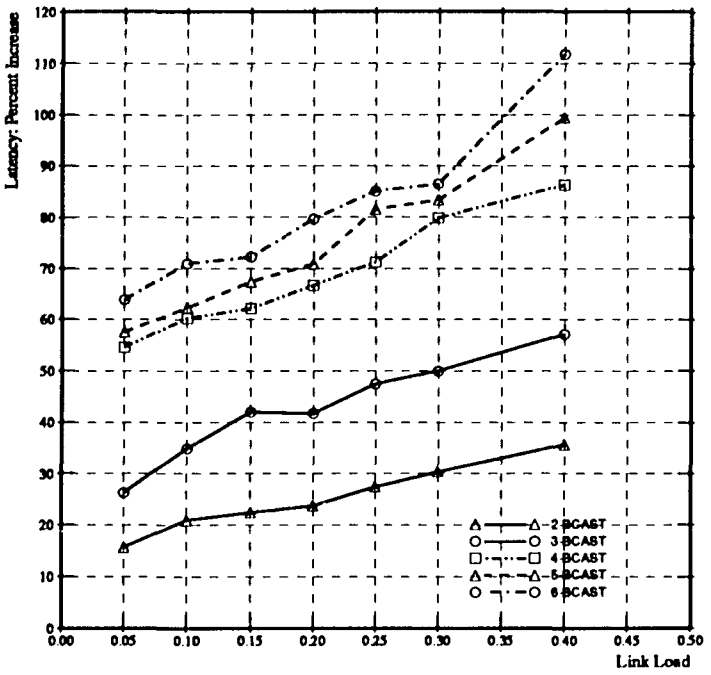

Figure 5.11: Performance of multiple copy broadcasts (mesh size = 7).

rate. For example, the routing hardware imposes a forced idle time corresponding to 8 bytes between the transmission of successive packets on any link. These access overheads decrease the effective throughput of the routing hardware, so saturation (for regular messages) sets in for link loads greater than 0.7. Since the broadcast message has to traverse multiple links, and the latency is determined by the last packet to be delivered, the latency increases rapidly with increasing link load.

Figure 5.11 shows the performance of the multiple copy broadcast algorithms compared to the SBCAST algorithm for a hexagonal mesh of size 7. The graph shows the percent increase in latency of the multiple copy broadcasts compared to the SBCAST latency for different link loads. It is observed that the ratio increases with increasing link load, showing that the latency of multiple copy broadcasts increases more rapidly than SBCAST. Part of the reason is that there are more messages to be delivered, and the latency is determined by the slowest message. The cost of 4, 5, and 6 copy broadcasts is quite close, because these three algorithms all use the wrap links to deliver messages and have a long path length.

The other objective of the simulation is to study quantitatively the benefits of using the virtual cut-through broadcast primitive. This is achieved by comparing the performance of our SBCAST algorithm against an algorithm which uses only store-and-forward packet

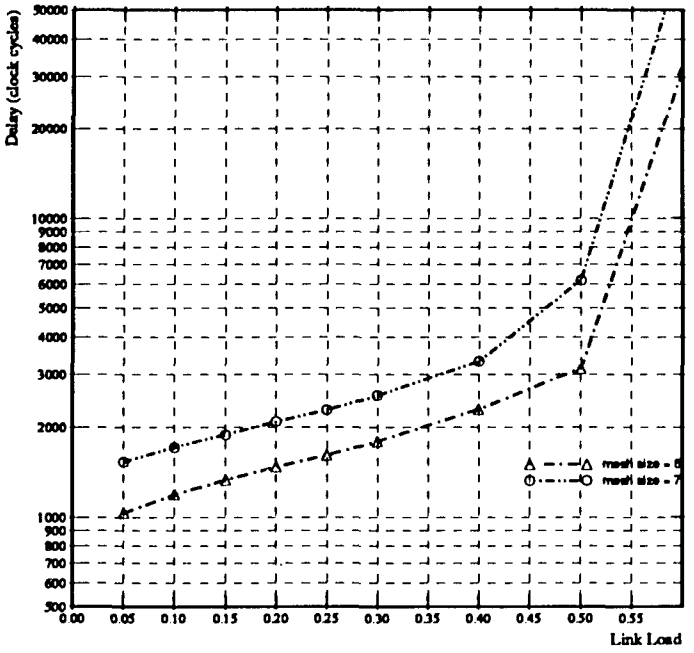

Figure 5.12: Performance of store and forward simple broadcast.

switching. This algorithm, termed SFBCAST, uses the same broadcast tree as SBCAST (see Figure 5.1). Two performance metrics are used for the comparison: one is *latency*, and the other is *mean delivery time* which is defined to be the average of the delivery time of a broadcast messages over all the nodes in the network. This second metric is necessary because latency, which is defined to be the delivery time for the last packet in the broadcast, does not consider the shorter delivery times of other packets and may be skewed by large queueing delays on some path. It is noted that the simulator does not count processing overhead in the nodes, but this tends to favor SFBCAST since packets which cut through do not incur the processing overhead in any case. This means that in practice the SBCAST algorithm would perform better, when compared to SFBCAST, than what is shown in the simulations.

Figure 5.12 shows the latency of the SFBCAST algorithm as a function of the link load, for meshes of size 5 and 7. The latency of SFBCAST is substantially higher than that of SBCAST for low link loads, but it increases more gradually with the link load. This can be clearly seen in Figure 5.13, which plots latency of the two algorithms for a mesh of size 5. Figures 5.14 and 5.15 show the effects of varying the mesh size on the relative performance of SBCAST and SFBCAST. Figure 5.14 plots the percent increase in latency of SFBCAST over SBCAST for different link loads and mesh sizes. This figure shows that

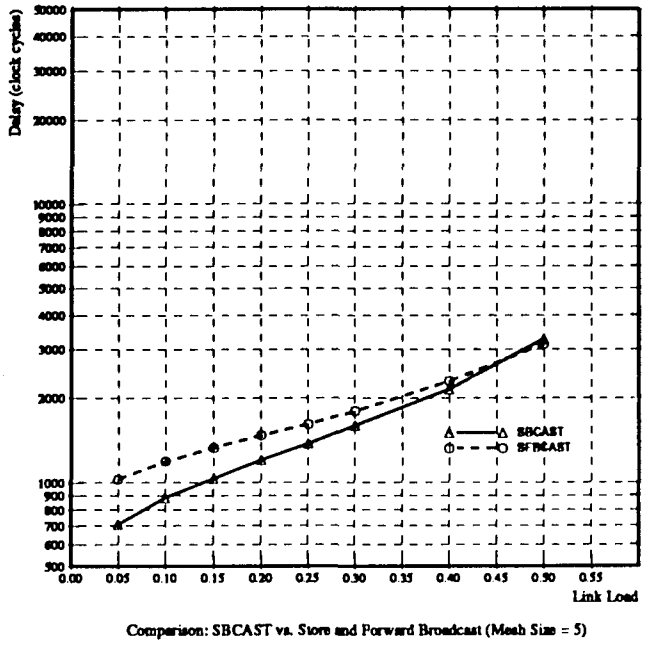

Figure 5.13: Performance comparison of simple broadcast algorithms.

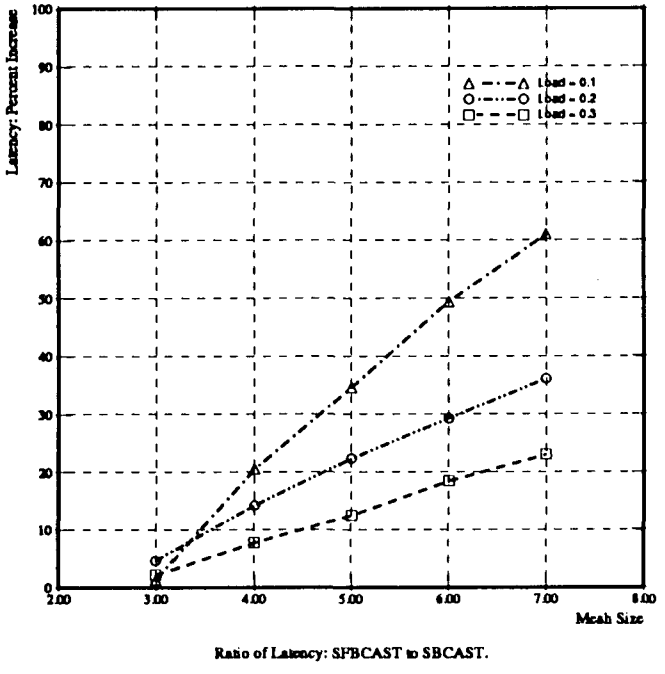

Figure 5.14: Comparison of SBCAST and SFBCAST with varying mesh size.

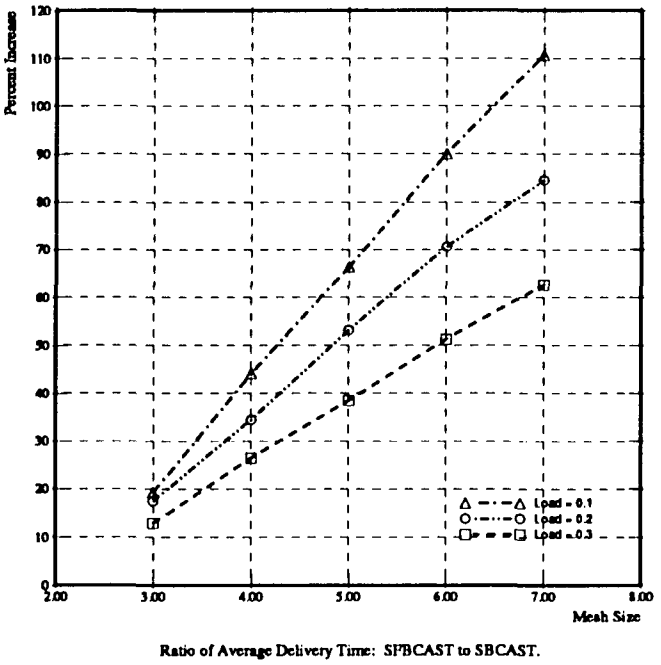

Figure 5.15: Average delivery time comparison with varying mesh size.

the difference in latency reduces as the link load increases, which is expected because the number of packets which cut through reduces with increasing link load. For a fixed link load, the difference in latency increases with increases in mesh size. The results of comparing the average delivery time (Figure 5.15) are similar, except that the percent difference is substantially higher in this case. These results reflect the advantages of virtual cut-through switching over store-and-forward packet switching.

5.6 Concluding Remarks

In this chapter, we developed a broadcast primitive which is applicable to interconnection networks with virtual cut-through switching. The primitive is well-suited for broadcasting in mesh-connected multi-computers, where a simple broadcast can be achieved using this primitive with only two message transmissions in the best-case. We also presented a family of broadcast algorithms based on this primitive, which deliver k ($k = 1, \ldots, 6$) copies of a message through node-disjoint paths to each node in the hexagonal mesh. The salient features of these algorithms are simplicity and the efficient use of virtual cut-through. The algorithms are particularly relevant to real-time systems, where the time overhead of identifying all the faulty processors on-line cannot be tolerated. Although the algorithms have been described separately, the relay procedures for the six algorithms can be combined into

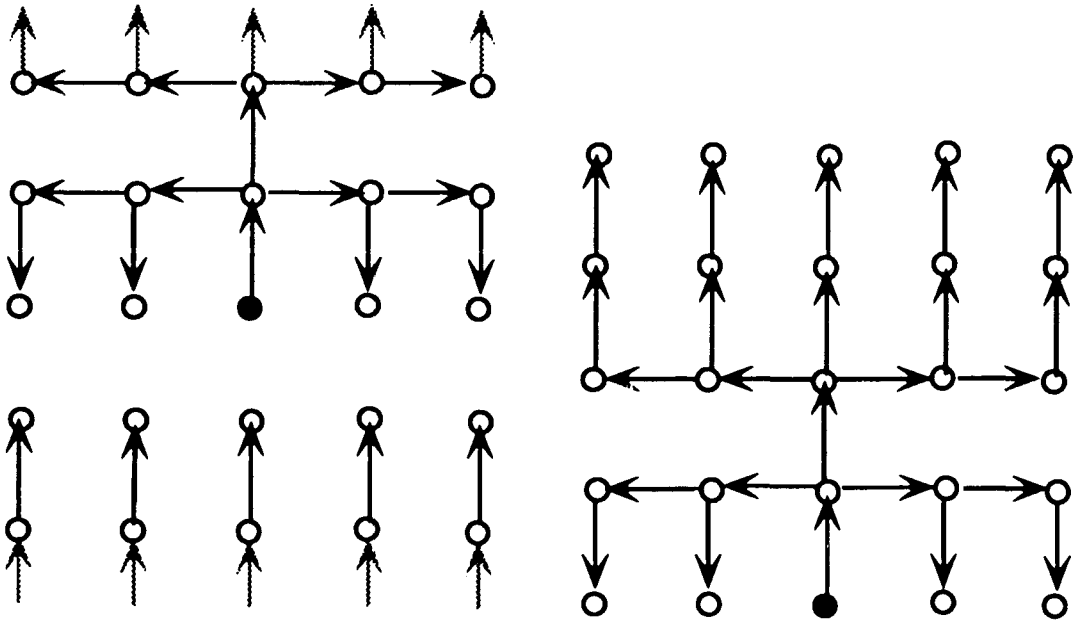

Figure 5.16: Broadcast tree for a wrapped rectangular mesh (in one direction).

a single function which makes decisions based on the type of broadcast in progress. We note that similar algorithms can also be developed for wrapped rectangular meshes. The broadcast tree for a 4-reliable broadcast algorithm in a wrapped rectangular mesh is shown in Figure 5.16. It can be seen that this tree is similar in structure to the 6-BCAST tree in Figure 5.6.

In HARTS, the RELAY primitive has been incorporated into the VLSI routing controller chip. The implementation of these broadcast algorithms on the HARTS network processor is discussed in Chapter 6. The NP has hardware support for time-stamping messages, and one immediate application of reliable broadcasting will be the establishment of a global time-base for HARTS, based on the clock synchronization scheme described in [RKS90].

CHAPTER 6

IMPLEMENTATION ON HARTS

In this chapter we describe the detailed design and implementation of the communication subsystem for HARTS, which was outlined in Section 2.1. One of the distinctive features of this subsystem is its use of the HARTS NP. The dedicated NP offers several advantages, especially in a real-time system. It does most of the communication processing, so the processing power of the APs can be devoted to application tasks. The NP handles all packet-level operations, so the number of interrupts seen by the APs is limited. It also results in a clear separation of scheduling domains: application tasks which run on the APs can be scheduled using their priorities and deadlines, while message processing tasks can be scheduled on the NP according to message priorities and deadlines. This avoids problems associated with determining the priority in case of conflicts, like when a high-priority task sends a message with a low priority. A dedicated NP may add to the cost of the system, but it can also be used to perform functions related to monitoring and system diagnosis. One possible drawback of this architecture is that the latency for application-level communication can increase due to the added level of indirection. However, this does not necessarily translate into longer execution time because of the execution-communication overlap between the APs and the NP.

Other systems have also used communication processors to improve network performance. The network adapter board of the VMP system [KC88] implements some link and network level protocols in the hardware. It is specifically designed for the VMTP protocol, and it provides a "device" interface to the application processors which are responsible for executing higher level protocols. The Nectar system [ABC+89] uses a communications adapter board (CAB) which features a general-purpose SPARC processor. The CAB software organization [CSRZ90] is similar to our design and provides a high-level interface to the host processors. However, it does not deal with the problem of real-time communication.

The NP is currently being developed by members of RTCL, and a block diagram is

shown in Figure 6.1. Its main components are the programmable routing controller (PRC), the network interface unit (NI), and the interface management unit (IMU). The PRC, which is an ASIC, subsumes the functions of the original routing controller [DRS89] and it also interfaces with the general-purpose IMU (a MIPS R3000 processor). The network interface consists of AMD's TAXI serial transmitter/receiver communication devices which are capable of achieving a data throughput rate of 100 Mbps.

The communication subsystem described in this chapter has, however, been implemented on the ENP-10 Ethernet processor. As mentioned earlier, the Ethernet is not a part of the HARTS architecture, but it serves as the system interconnect while the hexagonal mesh network is being developed. The ENP offers some of the functionality of the NP of HARTS, so we use it as a test-bed for software which is ultimately targeted for the real NP. For example, the ENP has an AMD LANCE Ethernet controller whose function is similar to the PRC. Thus, in the discussion that follows, we take the liberty of using the terms "NP" and "ENP" interchangeably.

In this chapter, we first describe the kernel that we have used on the NP in Section 6.1. This is followed by a description of the AP-NP interface and the view that is presented to processes running on the APs. We will then present the communication services provided by the NP, including a description of the packet structures used in Section 6.3. This section also discusses the implementation of the clock synchronization and reliable broadcasting algorithms. Section 6.4 deals with the implementation of real-time channels and the functions of the network manager. It also addresses the problem of error control and fault-tolerance for real-time channels. Lastly, in Section 6.5, we discuss the current status of the subsystem and provide some preliminary performance figures.

6.1 The NP Kernel

We have employed a derivative of the x-Kernel [HP91] as the executive for the NP, since it is well suited for supporting communication protocols. The x-Kernel provides several facilities for implementing protocols like a uniform protocol interface, and libraries to efficiently manipulate messages and maintain mappings. It also comes with utilities to configure and test different protocol stacks. However, several modifications are necessary to make it support the real-time communication service which are outlined in Section 6.4.1. The following is a brief overview of this kernel, summarized from [HP91].

The x-Kernel supports three types of communication objects: *protocols*, *sessions*, and *messages*. Protocols are static, passive objects and each protocol object corresponds to

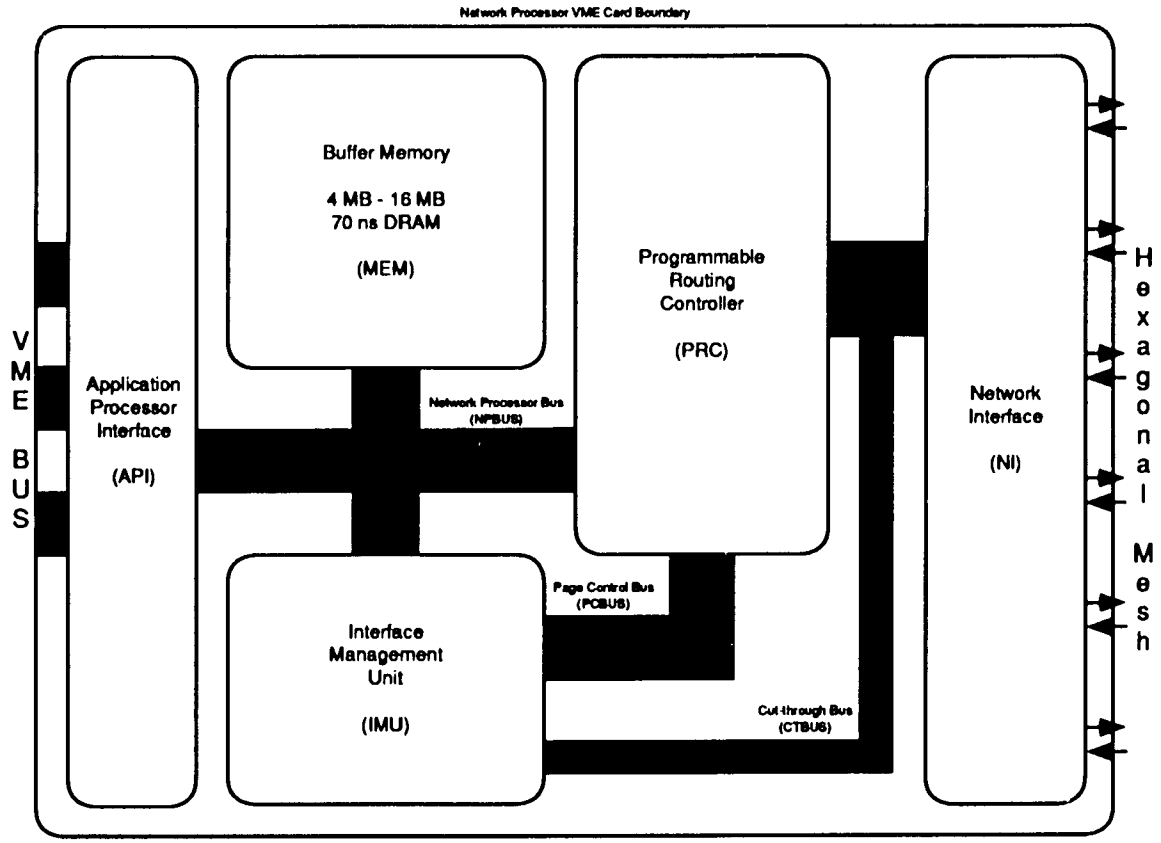

Figure 6.1: Network processor architecture.

Protocol Operations	session = x_open(hlp, llp, participant_set) x_openenable(hlp, llp, participant_set) session = x_opendone(hlp, llp, participant_set) x_demux(protocol, message)
Session Operations	x_push(session, message) x_pop(session, message) x_close(session)
Common	x_control(s, opcode, buf, len)

Table 6.1: Some Uniform Protocol Interface (UPI) operations.

a conventional network protocol. The relationships between protocols are expressed in the form of a protocol graph, which is compiled when the kernel is configured. Session objects are passive, but they are created dynamically. They represent the dynamic interaction between protocols and the data structures associated with them contain the local state of network connections. A message is an active object that is shepherded through session and protocol objects by a kernel process. Its representation consists of a stack containing protocol headers and a tree structure containing the user data.

The x–Kernel provides a uniform interface for interactions between protocols, some of the important operations on these objects are summarized in Table 6.1. The `x_open()` call is used by a higher-level protocol (hlp) to actively open a session with a lower-level protocol (llp), in order to establish a connection to some peer(s) specified in the participant set. This is typically used by the client side of a client–server pair. On the server side, the `x_openenable()` call is used by the higher-level protocol to look for connections on an address specified in the participant set. The lower-level protocol announces the establishment of a connection using the `x_opendone` upcall. The `x_demux` operation is used by a protocol to direct an incoming message to an appropriate higher-level protocol. `x_push` and `x_pop` are used to send and receive messages on existing sessions, while `x_control` can be used to perform various protocol specific control operations on protocols and sessions.

The message library provides numerous useful operations for handling messages. These include functions for fragmenting messages into packets and coalescing packets into messages, appending and removing (protocol) headers, etc. The map library maintains mappings and it is used primarily to map between session identifiers and session pointers. For example, a lower level protocol can use the port number field in a message (and other information) to find the session for which the message is destined.

Message Library	msg_make_new, msg_peek msg_break, msg_join msg_push, msg_pop
Map Library	map_create map_bind, map_resolve

Table 6.2: Some library functions.

Port to the ENP

The kernel running on the ENP is based on version 3.0 of the x-Kernel for the Sun-3 platform. The x-Kernel sources for the Sun-3 were mostly written in C, with a few low-level routines in 68020 assembly language. Some of the C source files had 68020 dependencies, like the routines which manipulate the process stacks and the exception frames. The changes required for the ENP implementation can be attributed to (a) differences between the 68020 and 68000 architecture and instruction set and (b) differences between the Sun-3 and ENP-10 designs. The first category affects things like process management and exception handling, where exception stack manipulation is required. Differences in the second category affect the memory management, interrupt and exception processing, and the I/O facilities.

The ENP-10 does not have any memory mapping hardware, nor does it make any distinction between user and supervisor state for memory access. Therefore, the process management routines have been modified to remove all virtual address space support. Also, since we do not intend to support user processes on the ENP, we have not provided any support for them. All processes running on the ENP are kernel processes, which handle communication-related tasks for the processes running on the host processors or APs. The User system call trap handlers and the user interface routines of the Sun-3 version have therefore been left out. In their place, however, we have the interface between the host processors and the ENP. Further details of this port can be found in [Kan].

6.2 The Application Interface

The application interface is split between the APs and the NP, and it has been designed to provide both synchronous and asynchronous forms of message delivery. Communication between the ENP and the host processors is accomplished using mailboxes in memory, and inter-processor interrupts. Host processors can interrupt the ENP by access-

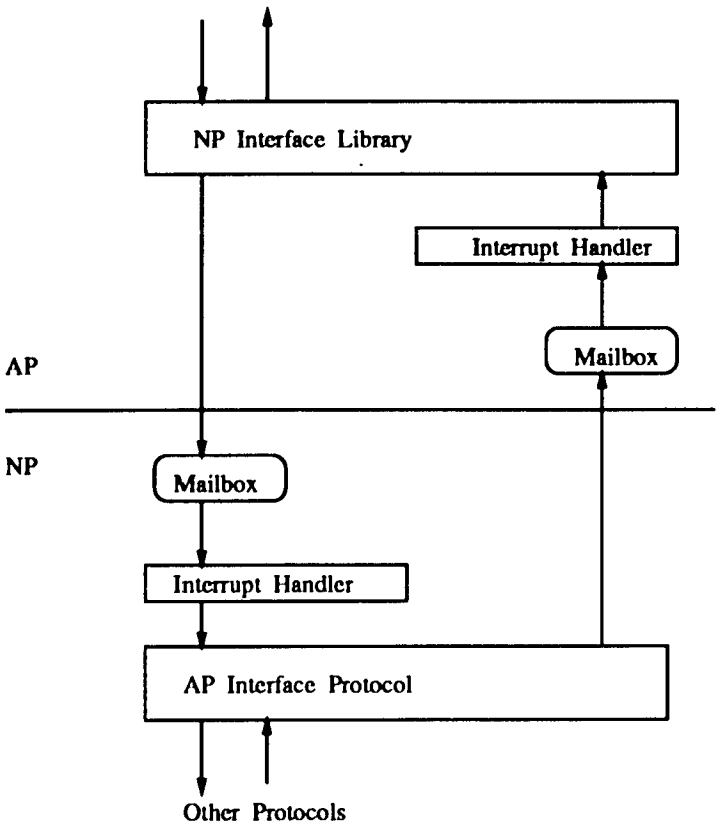

Figure 6.2: The host interface.

ing any location in a certain page of the ENP memory, but the access itself is dummy. Host processors also support a similar interrupt mechanism, so the ENP can interrupt any host processor. Figure 6.2 shows a block diagram of the interface.

There are two mailboxes for every AP, one each for communication in either direction, so that a mailbox is accessed by exactly one producer and one consumer processor. Each mailbox is organized as a circular buffer with a fixed number of slots, and head and tail pointers. The producer processor can only modify the tail pointer, while the consumer processor can modify only the head pointer. Hence, we are able to achieve consistent access to the mailbox without using read-modify-write operations[1]. A mailbox slot consists of three elements: an operation code, an argument pointer, and a priority number. The normal operation on the part of the consumer is to map the operation code into a handler procedure, and pass the argument pointer as a parameter to this handler. While passing pointers between the host processors and the ENP, it is necessary to translate the addresses, because

[1]Read-modify-write operations can result in bus exceptions on the 68000 which are not easy to handle.

the ENP address space is different from the host address space. In our implementation, all such translation is done in the ENP and the host is not aware of this operation.

6.2.1 Host Side Interface

The User Interface allows pSOS processes running on the host processors to access the communication facilities of the ENP. In addition to providing the asynchronous receive mechanism supported by the x-Kernel, we also provide a synchronous receive which is similar to the pSOS IPC primitives. Access to the NP services is accomplished by:
(1) creating a host protocol object,
(2) linking this to the appropriate NP protocol object to obtain a session, and
(3) invoking operations on the session.

The host protocol object is required for any operation with the NP, since it contains structures used for the AP-NP interactions. It is created using `xcreateprotl()` and the object is located in host memory. This operation initializes the protocol object state and creates a pSOS agent process which is used to handle upcalls from the NP. It then signals the create protocol operation on the NP which is responsible for further state initialization. The *state* associated with this object is shown in Table 6.3. It contains three types of structures: structures used exclusively by the host processor, like the session structures through which application tasks receive messages; structures used to convey information between the host and the ENP, like the send-reply mailboxes and the upcall request-reply mailboxes; and structures used only by the ENP, like the interface protocol entry points and the receive buffers.

The host protocol can be linked to an NP protocol using the standard **xopen** and **xopenenable** functions. The **xopen** call returns a host session structure which is used for further message send and receive operations. This structure contains pointers to the host protocol object, the NP session object, and semaphores for ensuring exclusive access. The list of all available User Interface functions, and their description, is presented in Appendix 6.B.

On the receive side, since pSOS does not provide an asynchronous signal mechanism, we use the pSOS agent process which is attached to each protocol object to handle upcalls. The handling of the upcall depends upon the parameters used for the protocol. For example, consider the **demux** upcall. The agent process first locates a host-specific session structure corresponding to the ENP session on which the message arrived. If a host session is not found, as in the case of the first message, a new host session is created and initialized.

```
int         *send_mutex;          /* host related stuff */
int         *reply_sem;
int         *upcall;
int         agent_pid;
int         priority;
buf_hdr     *apbuf_list;
int         *accept_wait;         /* to wait for new sessions */
u_long      new_sessns;           /* circular list */
PFI         demux;                /* UPI procedures */
PFI         opendone;
PFI         closedone;
PFI         control;
u_long      sendbox[5];           /* AP - NP parameter area */
u_long      replybox[4];
u_long      upcallbox[5];
u_long      upcallreplybox[4];
buf_hdr     *buf_slot;
short       cpu;                  /* NP related stuff */
int         *upcall_mutex;
int         *upcall_reply;
buf_hdr     *npbuf_list;
```

Table 6.3: Host protocol object state.

New sessions are inserted in a queue in the protocol state for subsequent presentation to application processes. If there is a process waiting for a session, it is awakened.

After the host session has been identified, message delivery can be synchronous or asynchronous. If the host protocol has the **demux** procedure defined, the agent process invokes this routine and passes it the received message as a parameter. Otherwise, it inserts this buffer into a receive queue which is a part of the session structure. If there is any process waiting for messages, the agent performs a signal operation to awaken the process. Message retrieval from the session is based on the queueing discipline specified for the session, the default being FIFO.

6.2.2 ENP to Host Interface

The ENP side interface has been written as a protocol module, so that it can be configured easily into a protocol stack using the *x*-Kernel protocol configuration tools. This makes it possible for us to construct two types of applications: those which only use the ENP processor, and those which use both the host and the ENP processor. The former type is especially useful for testing new protocols, while the latter is the standard configuration for applications.

The mailbox interrupt handler is set up as part of the protocol initialization procedure. The interrupt handler extracts the contents of a mailbox slot and interprets the requested operation. In most cases, it creates a new kernel process with the requested priority level to handle the requested operation and passes the argument pointer to it. The operations supported are the standard *x*-Kernel Uniform Protocol Interface (UPI) operations, with a couple of additional operations:

findprotl:	to find a protocol object given its name, and
req_v:	to signal a semaphore object.

All operations other than **req_v** accept a protocol object as the argument. The *state* associated with this protocol object contains the actual parameters for the operation, and other related control structures. The results of the execution are also returned to the requesting process on the host through the protocol state. The requesting process is assumed to be blocked, waiting for a reply, on a semaphore.

Among the important UPI operations, **createprotl** is used to initialize the ENP—related structures in the protocol object. This includes initializing the protocol state, and the pointers to functions related to this object, like the **demux** function. The protocol object itself is created by the host process and it resides in the host memory. The **push** operation

creates a message structure and copies the packet data from the host buffer into the message buffer. It then pushes the message to the lower-level protocol on the appropriate session (that is another parameter). For long messages, that is, those which would require multiple packets for transmission, the message structure is created but the copy operation is deferred. The copying would take place, on a packet-by-packet basis, only when the message is ready for transmission.

On the receive side, the interface provides routines like **demux** and **opendone** which handle received messages on behalf of all host protocol objects. These routines pass parameters to the host using the upcall boxes in the protocol state. Access to these boxes is serialized using a semaphore. After signalling the appropriate host processor, the routines block for a reply using a semaphore.

Messages arriving at the ENP reside in local memory. The message contents have to be copied into the host memory before the host is signalled. For this, the **demux** routine must have access to receive buffers in the host memory. We handle this problem by forcing the host to provide receive buffers for the protocol object. These receive buffers are actually allocated by processes on the host which use the protocol object, and their size is determined by the application.

6.3 Communication Services

Communication services are provided in the form of protocols running on the *x*-Kernel. Fig. 6.3 gives an overview of the communication subsystem, and shows the dependencies between various protocols. The link-level protocol supports broadcast addresses, and also provides a reliable broadcast mechanism. The clock synchronization protocol maintains a system-wide synchronized time base, which can be accessed by application tasks, and by other protocol modules. Other services provided include a name service, user datagram service, a request-reply service, etc. The request-reply service (i.e., RPC) is used by many protocol modules, including the name service and the real-time channel service. Before describing these protocols, we will first present the packet format that we have developed for the HARTS.

Figure 6.3 also shows several protocol modules related to the real-time channel (RTC) service which will be described later in Section 6.4. The RTC link level has been made distinct from the normal link level mainly because of the differences in buffer management and in-transit message handling. The dashed-line block in Fig. 6.3 indicates that the Network Manager, which is a centralized service that handles RTC establishment and

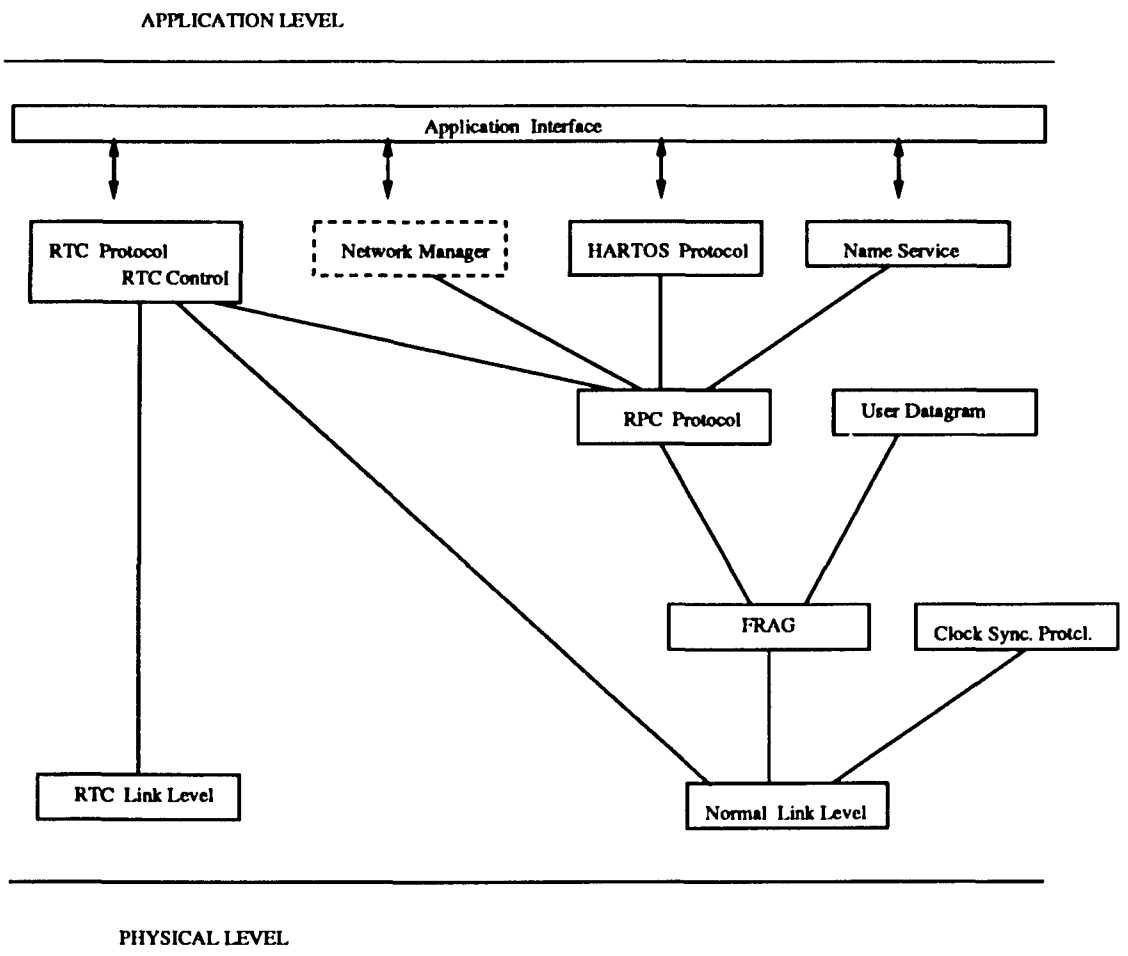

Figure 6.3: Simplified view of the communication subsystem.

maintenance, is not present on all nodes in the system.

6.3.1 Packet Types

The hexagonal mesh packet format has been derived after considering the different types of messages that are supported, and the hardware support that they require from the HARTS Programmable Routing Controller (PRC) at the link level. It represents information which is used by the PRC and determines the method used to handle the packet. The packet contains framing bytes, message flags, route specification, priority, and other fields. A description of the fields in the packet can be found in Appendix 6.A.

The route specification and some of the other fields depend upon the packet *type*, which is determined by the `flags` field in the packet header. There are three types of packets: normal, broadcast, and source route packet. The packet type also determines

protocol module to be used for handling the packet. For example, a packet with the RTC flag is handed over to the RTC Link Protocol. The source and destination fields are 16-bit **hnet** addresses, and the different types of broadcasts have been assigned special addresses.

Normal Packet: None of the packet flags is set for this type of packet. The destination field is specified using three signed **Routing Tags** which give the distance to the destination node in six directions. The PRC uses the hexagonal mesh dynamic routing algorithm which was described in Section 5.2 to process packets of this type. Route specification:

```
Routing Tag x (1 byte)
Routing Tag y (1 byte)
Routing Tag z (1 byte)
```

Broadcast Packet: The Broadcast flag is set for packets of this type. The PRC treats the first byte of the Route Specification as a **Hop Count**. It decrements this count and if the count is non-zero, it tries to propagate the packet forward in the direction in which it is traveling. Moreover, a copy of any Broadcast packet is automatically dropped off to the node. The type of broadcast, whether simple or k-copy, is indicated by the destination address. The other two bytes of the route specification, tag and step, are the state information required to implement the broadcast algorithms of Chapter 5. This information is used by the Link Protocol to forward the packet appropriately to other nodes. Route specification:

```
Hop Count  (1 byte)
Tag (1 byte)
Step (1 byte)
```

Source Route Packet: The Source Route flag is set for this type of packet. It is envisaged that this type of packet will be used mainly for real-time channel traffic, so the RTC flag would probably also be set. The destination node is specified in the form of a route leading to it. The route is specified in terms of a sequence of links to be traversed. To specify 1 out of 6 links, a minimum of 3 bits are necessary; we have chosen to represent this using 4 bits, a nibble. The PRC would pick up the next link to be traversed from the Route field and fill that nibble with zeros. When the Route field is all zeros, the PRC can recognize that the packet has reached its destination. Route specification:

```
Route to destination, 1 nibble per hop
   (fixed size: 3 bytes, gives a max. path length of 6)
```

6.3.2 Link Level Protocol (LLP)

This protocol sits on top of the device driver and handles both Normal and Broadcast packets. Its clients are different protocols, like the Clock Synchronization Protocol, and it demultiplexes messages to the different client protocols based on the Protocol Number in the packet. It uses the Priority field in the packet to determine the order of service for the packets.

Different types of broadcast operations are supported: from simple broadcasts to 6-reliable broadcasts, and the type of broadcast is be determined by the destination field in the packet. Broadcast packets also carry some additional information in the route specification field, as described earlier. When a broadcast packet is received, this protocol module notes the direction in which it was received and checks to see whether the packet was able to propagate forward using virtual cut-through at the PRC. If not, it assumes the responsibility of propagating the packet forward in conformance with the RELAY primitive. It then uses the destination, hop count, tag, and step fields, and the direction information, to determine whether the packet has to be forwarded in any other directions.

Broadcast communication can be made more efficient if it is possible to send a packet on multiple (preferably all six) links simultaneously. Although the data portion of the packet would be the same, the header information differs, depending upon the direction. In this case, it is highly beneficial to have the ability to make different headers point to the same packet data. The PRC makes this possible by allowing packets to reside in a chain of buffers, where the first buffer contains the link packet header. On the transmit side, the LLP places the header in a separate buffer and the rest of the packet in one or more packet buffers. This packet header is of fixed size and, on reception, it is again placed in a separate buffer by the PRC. This mechanism allows intermediate nodes which have to forward the broadcast packet in different directions to function efficiently, since they do not have to make copies of the data part of the packet.

One simple client of the this protocol is the User Datagram Protocol (UDP), which provides datagram service to user processes. It provides ports, to which processes can attach to send and receive messages. The UDP can directly access the LLP for short messages, and it can use the FRAG protocol to handle messages that are longer than the maximum packet size. The packet format for this and other protocols can be found in Appendix 6.A.

6.3.3 Clock Synchronization Protocol

The clock synchronization protocol (CP) module interacts with its peers on other nodes to provide a global time-base for the system. It relies on a hardware timestamping mechanism to disseminate its clock value to other nodes [RKS90] and uses the Interactive Convergence Algorithm[LMS85] on the clock values that it receives from other nodes. The PRC affixes a transmit timestamp to a clock packet just before its transmission and it appends a receive timestamp to any clock packet that it receives. This hardware timestamping ensures that delays in the processing and propagation of clock messages can be factored out and they do not affect the tightness of the synchronization. The protocol also employs a hardware maintained local clock, which is 64-bit wide and has a resolution of 1 microsecond.

To implement interactive convergence, the CP must transmit and receive clock messages, which it accomplishes using the LLP. It sits on top of the LLP and binds itself at a special protocol number, so that any packets received with that protocol number are forwarded to it by the LLP. It transmits packets using the LLP, with the destination host specified to be a 5-reliable broadcast address. The CP requires access to the Clock flag in the link packet header, so the LLP provides a control operation for this purpose. The LLP also assists the CP in handling clock packets that pass through the node as a part of the reliable broadcast. For these packets, it adjusts the time-stamp fields as specified by the clock synchronization algorithm before propagating the packet to other nodes.

The clock maintained by this protocol can be read directly by processes running on the NP. It is used to set deadlines for messages and processes, and to determine the order of service. The protocol also provides control operations using which processes on the APs can access the time.

6.3.4 Remote Procedure Call

This protocol implements a request–reply operation with *at most once* semantics, using the technique of Birrell and Nelson [BN84]. The protocol handles only the RPC transport mechanism. Clients of this protocol are expected to marshal the call arguments into a request packet, and subsequently extract the return results from the reply packet.

We can implement the existing HARTOS calls [KKS89] in the form of a client protocol module sitting on top of RPC. Some of the HARTOS calls are not necessary because their function has been subsumed in other protocols. For example, the name service calls are handled by the Name Service Protocol. Similarly, the data transfer operations can trivially be implemented using UDP and the fragmentation protocol, FRAG (below).

6.3.5 Fragment Protocol (FRAG)

This protocol fragments large messages into link level size packets and transports them. On reception, it collects the fragments and coalesces them into a single message. At present, it is designed to handle a maximum of 32 link level size packets. The protocol is unreliable and does not guarantee successful message delivery. It is a blast protocol based on the Sprite RPC's fragmentation algorithm. Currently, it is used to transport both RPC and UDP packets.

6.3.6 Name Service

The name service provides a mechanism for associating names to objects and for locating objects by their name. The service can be used for locating port numbers, processes, etc., by application processes and by other services on the NP. It is implemented as a client of the RPC protocol, which it uses to talk to its peers on other nodes. The protocol uses broadcasting to locate remote objects, so the RPC protocol was modified to accept broadcast destination addresses.

6.4 Supporting Real–time Channels

The real-time channel service is provided by several modules running on the NP which handle the different phases of a real–time channel, i.e., message transmission and channel establishment. The Real–time Channel Protocol (RTCP) provides the front-end for all user–level operations related to real–time channels. It accepts user requests for channel creation/deletion, and message transmission on existing channels. It interacts closely with the RTC link protocol, which among other things handles transit packets belonging to real–time channels.

It has been shown that channel establishment is a complex operation and it involves reservation of resources at multiple nodes in the network. It is therefore preferable to place the channel establishment function into a separate service. By making this function centralized, it is possible to make better use of network resources since we can select routes appropriately to balance network load. This approach also makes it easier to handle network reconfiguration in the event of network failures. This service, called the *Network Manager* (NM), is provided by a special node (or a set of nodes) in the system. We now describe these protocols in detail, starting with the RTCP.

6.4.1 Real–time Channel Protocol

This protocol implements the real–time channel communication scheme and it handles the different types of messages described in Section 2.2.1. The clients of this protocol are processes running on the APs. It handles the transmission and reception of messages of varying sizes. As defined, a real-time channel is a uni-directional connection between two end-points. In order to support bi-directional communication, the RTCP can be extended to allow the user to create a pair of real–time channels simultaneously. The parameters for these two channels would be specified separately, as they are expected to be different. The RTCP module consists of two logical parts: a send–receive part and a control part (RTCC).

Channel Control

The control part handles requests for channel creation and teardown, and it interfaces with the NM using the services of RPC as shown in Figure 6.3. It also talks with its peer RTCCs on other nodes in the network to accomplish these functions. The actions taken depend upon the type of message that is specified. The description of the handling of Alert and Best-effort messages is deferred to the end of this subsection. In the case of real-time message channels, central reservation is required because these messages require delivery time guarantees. This delivery time is measured from the time the request is presented to the NP to the time it is delivered to the AP on a remote node. The RTCP is responsible for transferring data to and from the local memory of the APs and the overhead incurred for this (and other functions) has to be accounted for in the computation of the end–to–end delivery time. Hence, the delivery time computation includes processing and buffer–copying time at the source and destination nodes, and message transmission time on each link of the route. It does not, and cannot, include scheduling delays at the receiving AP because these are determined by the application task priorities and the scheduler on the AP.

Channel establishment proceeds in two phases. In the first phase, the control part frames the parameters of the channel into a create–request message and makes a remote procedure call to the NM. The request and reply packets used for interaction with the NM for channel creation are shown below. If the channel creation request is successful, the NM returns a message containing the selected route for the channel, and the worst-case delays for each link on the route. In the second phase, the RTCC again uses a series of remote procedure calls to forward this information to each node along the route to the destination. This two-phase scheme tries to split the task of channel creation between the requester and

the NM, to reduce the load on the NM.

```
/*
 * Channel Establish Request
 */
struct est_reqst {
     hnet_addr source;
     hnet_addr destn;
     int  msg_size;
     int  msg_inter_arrival;
     int  burst_size;
     int  requested_delay;
};
```

```
/*
 * Channel Establish Reply
 */
struct est_reply {
     enum {FAILURE, SUCCESS} status;
     int  channel_id;
     int  total_delay;
     int  num_links;
     struct link_info {
         int node_num;
         int link_num;
         int link_delay;
         int cumulative_delay;
         int horizon;
     } links[ROUTE_MAX_LINKS];
};
```

The channel creation information is received by the corresponding RTCC modules and recorded in data structures which are used to set deadlines for messages belonging to real-time channels. If a failure occurs during this phase of operations, the RTCC tries to close the channel and free up the resources. Otherwise, a session is created corresponding to the new channel and a server process is created to service the session. The function of this server process will be explained in the following section. The session pointer is returned to the requesting task and this has to be used in all subsequent operations on the channel. On the destination node, the RTCC performs a similar operation to create a session.

To close a channel, the RTCC only has to inform the destination node and the NM. The destination node has to be informed so that the receiving end of the channel can be notified. However, the intermediate nodes do not have to be informed because the NM keeps track of all state information regarding transient load. To clean up the state associated with the channel, the RTCC has to destroy the session and the server process associated with it.

Kernel Modifications

The run-time environment provided by the x-Kernel is not adequate to support real-time channels. Several modifications and enhancements are required, mainly in the areas of process scheduling and buffer management, as described below.

Scheduling: The x-Kernel uses a fixed-priority scheduling policy, where processes in kernel mode are not preempted. We need a deadline-based scheduling scheme with multiple classes of processes, similar to the message scheduling policy described in Section 3.2.3. We classify active processes into three queues: high-priority real-time processes, non real-time processes, and low-priority real-time processes. Processes now have deadline and arrival time attributes associated with them, derived from the messages that they handle. Also, to preserve the message scheduling model, the network device driver has to maintain queues for different types of packets. (This is technically not a part of the kernel.)

Processes: The number of processes that can be created is limited by the available kernel memory because each process requires a private stack. Therefore, the process-per-message model could result in a loss of messages if processes are not available to handle them. In HARTS, there are two factors which can increase the demand for processes: packets in transit, and real-time channels. Real-time channels pose a problem because messages which arrive early (in Queue 3) may not be eligible for processing. If a process were used to handle each message, these messages would tie up processes. We can handle in-transit packets by letting the receive interrupt handler take care of the forwarding. Our solution to the second problem is to adopt a process-per-channel policy for real-time channels. This is sufficient because messages belonging to the same channel have to be processed in the order of their arrival. To support server processes of this type, we require primitives to suspend and resume processes with a certain priority/deadline, operations which are more efficient than process creation.

Buffer Management: Currently, buffers are allocated from a common pool when the message is created. The buffers are released when the message is destroyed after transmission, provided there are no other outstanding copies of the message. (The message data structure contains reference counts to keep track of the copies.) A common buffer pool can create problems for real-time messages because demands for buffers from non real-time messages could use up the pool. In order to reserve buffer space for real-time messages, it is necessary to maintain a separate pool of buffers for these messages. To achieve this, we have to tag the buffers, so that they can be returned to the appropriate buffer pool when freed. On the NP, one possible mechanism is to use the address range to identify real-time packet buffers.

The receive side deserves special attention. A pool of receive buffers has to be provided to the PRC to receive incoming packets. In order to preserve buffer space for

real-time messages and prevent buffer overruns, this pool has to be replenished whenever a packet is received. The packet should be accepted only if a free buffer is available, which in turn depends upon the type of the packet. For example, if the packet belongs to a real-time channel, a new receive buffer has to be allocated from the pool of buffers reserved for real-time channels.

Message Transmission

RTCP treats each send request to be a separate message. It handles messages of varying sizes and in the case of large messages, which are larger than the maximum link packet size, it does fragmentation and reassembly. However, in the usual case of real-time messages, RTCP does not expect any acknowledgment, nor does it attempt to retransmit packets. The real-time channel send request, which is a push operation on the session, has to be treated as a special case in the application interface. Instead of creating a process to handle the request, the interrupt handler directly invokes a special send function in the RTCP. This procedure first checks whether the send request satisfies the constraints of the arrival process for the channel. It rejects the send request if these constraints are violated so as to prevent the channel from exceeding its allotted resources. Otherwise, it computes the logical arrival time for the request and queues the request for service. The operation is non-blocking and the procedure exits with a return code that indicates the time by which the message is guaranteed to be delivered. In the case of a rejected request, the sending process can detect the error condition from the return code and take appropriate recovery actions. For example, it can resubmit the request at a later time, or send it as a best-effort message.

The channel server processes the requests in order of their arrival. The deadline of this server process is determined by the request that it services. It has to copy the message in from AP memory and create packets for transmission. It places these packets into the appropriate queue in the device driver. On the destination node, the channel server collects the fragments of a message. The priority of this server is again determined by the deadline of the message that it services. It copies the message into a receive buffer associated with the channel, provided by the application task. If a receive buffer is not available, the message is immediately discarded so that it does not tie up buffer space in the NP. If some fragments of a message do not arrive within the deadline for the message, the partial message will be delivered to the client with an appropriate warning.

Other Message Types

We now describe how to handle the other types of real-time channels, and some special cases of real-time message channels. In the case of Alert messages, the RTCC has to interact with the NM to find appropriate disjoint routes. Alert messages are treated as single packet messages whose inter-arrival time is infinite. The NM selects multiple routes for the channel as specified in the reliability parameter. In the selection process, it checks for contention with other Alert channels and tries to minimize the contention.

The reliability field can also be specified for real-time message channels. It is treated as a specification for error control. A non-zero value indicates that the RTCP has to enforce error control and try to compensate for packet loss. Since we reserve buffer space and bandwidth for real-time messages, transmission errors are the main remaining cause for packet loss. We use two different methods for error control, the choice is determined by the size of the message as specified in the channel parameters. These methods try to reduce the probability of message loss due to transient transmission errors. Network failures are handled separately by reconfiguring the channels that use the failed component.

Short messages are handled by transmitting two copies of the message over the same route at two different times, and the destination is responsible for choosing a copy from the packets that it receives. The advantage of this scheme is that it is simple and it does not require any acknowledgment from the receiver. Moreover, the channel establishment procedure is essentially unchanged, except that the message size is now doubled. The alternative method of selecting two node-disjoint routes for the channel would require several modifications to the channel establishment procedure, and it would also increase the overheads of channel establishment.

The technique used for short messages may not be appropriate for long messages because of the excess resource requirement. Hence, we use a selective retransmission scheme for long messages. If the receiver detects an error in a received packet, or a dropped packet, it sends a negative acknowledgment to the sender containing the identity of the lost packet(s). Retransmission occurs after all the packets in the message have been transmitted. It can be shown that selective retransmission of lost packets can significantly reduce the probability of message loss, even if there is a bound on the number of packets retransmitted. A bound on the number of retransmissions is necessary because of the timing constraints on the message. However, it can be shown that even if retransmission is restricted to a single packet the probability of message loss can be reduced significantly.

Assuming that the packet loss probability is p for any packet, the probability of

successful delivery, without retransmissions, for a message containing n packets is $(1-p)^n$. With a selective retransmission policy restricted to one packet, and assuming that the packet loss probability for the NACK and the retransmitted packet is also p, the probability of successful delivery is improved to $(1-p)^n + np(1-p)^{n-1}(1-p)^2$. For example, with a packet loss rate of 1 in 1000, the probability of successful delivery for messages of different sizes without retransmission, and with selective retransmission is summarized below.

No. of Packets	Prob. of Delivery	One Retransmission
8	0.9920	0.9999
16	0.9841	0.9998
32	0.9684	0.9994
64	0.9379	0.9979
128	0.8797	0.9922

The single packet retransmission policy has the added advantage that the time reserved for retransmissions is small. We therefore adopt this policy for transmission of long messages. We use best-effort delivery for the NACK and the retry packet because these packets are generated sporadically. This decision also allows us to retain the channel establishment procedure with slight modifications. The RTCC now requests the NM for a message deadline which is shorter than the actual deadline. The deadline is determined by considering the time required for exchanging the NACK and retry packets between source and destination. As shown in the figure below, the transmission time of the NACK and the retry packet overlaps with the time allotted for copying the packets into the AP memory at the destination.

Best-effort messages do not require any hard guarantees, so the RTCC does not have to request the NM for any reservations. If the reliability field is zero, the RTCC is free to use dynamic routing so the channel establishment is trivial. Otherwise, RTCC selects two disjoint routes to the destination using path length as the metric for the selection. We do not permit more than two copies for the best-effort channel in order to limit the run-time resources required for the channel.

6.4.2 Network Manager

The Network Manager handles requests for creation and deletion of channels. It maintains state information for all nodes in the system and implements the channel estab-

lishment scheme outlined in Section 3.2. The NM maintains several structures which are vital to the channel establishment procedure. It maintains a table containing the resource requirements and the assigned route for all existing channels in the system. For each node, it also maintains a structure containing information about the links emanating from the node, the channels which use these links and their relative priority, and the buffer space allocation. In order to ensure the consistency of this data, the NM serializes the channel creation/deletion operations.

The procedure used for channel establishment differs slightly from the one shown in Fig. 3.2. The processing time for a message at the source and the destination of a channel can be substantial because it includes time for copying the message into and out of NP buffer memory. In order to compute the worst-case end-to-end delivery time, it is essential to compute the worst-case response time at the source and destination nodes. This response time is affected by the processor scheduling policy employed on the NP. As described in the previous section, we have adopted a processor scheduling scheme in which we classify processes into three categories based on the type of message that they handle and assign deadlines to processes based on the message deadlines. This scheduling policy enables us to compute the worst-case response time for messages using the same technique that we use for computing the worst-case link delays. The actual processing time for a message is proportional to its length and since HARTS is homogeneous, it is not node dependent.

We employ the incremental routing algorithm, INC, described in Section 4.5 to select routes for channels. Since the worst-case link delay for a channel depends mainly on the other channels using that link, we do not have to consider other traffic while computing the link load. The algorithm for computing the worst-case link delay, which is based on the procedure in Appendix 3.A, is a pseudo-polynomial time algorithm. Its execution time depends on the number of channels and the inter-arrival time of messages on the channels. However, we expect this algorithm to be quite efficient in practice, especially because the route selection scheme tends to balance the load and reduces the number of channels assigned to a link. Our initial experience with the algorithm supports this assertion.

The buffer requirement for a channel over a link depends upon the assigned delay for the link, and it is computed in the final phase of the procedure. In each node, the buffer space available for real-time channels is partitioned into two parts: space reserved for channels which have their source on the node, and space for channels which pass through or terminate on the node (transient space). The transient buffer space is shared among the

links connected to the node. For each link, the buffer space required is computed based on the assigned delay for the channel at that link and the horizon for the link. If this requirement exceeds the available buffer space at the node, an attempt is made to find a reduced horizon which would satisfy the buffer constraints. Upon successful channel establishment, the (possibly new) link horizons are included in the reply message to be propagated subsequently to the nodes which form the route for the channel.

Fault-Tolerance

The current version of the NM is not replicated, so it is susceptible to single-point failure. To protect the system against single-point failure, it would be necessary to replicate the functions of the network manager on multiple nodes. However, this gives rise to the problem of maintaining consistency amongst these replicated tasks. We have to make sure that that the replicas of the network manager coordinate their activities to avoid inconsistencies.

We have developed a scheme for replication in which the network management functions are placed on three adjacent nodes, so that up to two failures can be tolerated. This is based on the reliable broadcast scheme developed by Chang and Maxemchuk [CM84]. One of the three network managers is picked to be the *token holder*. The client node broadcasts its request using a simple broadcast. The token holding manager assigns a serial number to the request and processes the request in sequence. Once a request is processed, the manager sends the reply to the other managers so that they can update their state information. It cannot send the reply back to the client until it is certain that at least one of the other managers has a copy of the reply. This precaution is necessary because the token holding manager could fail after sending the reply, which could leave the other managers in an inconsistent state. The token is made to circulate between the manager nodes to make sure that their state remains consistent.

In this scheme, the client node is responsible for retransmitting requests if it does not get a reply within its timeout duration. The clients and the managers use the request-reply protocol which automatically handles duplicate requests and replies. This scheme also has the advantage that it can be easily extended to accommodate more manager nodes.

Failure Handling: The failure handling scheme relies on monitoring, in which the NPs monitor each other to detect link or node failures. If a failure is detected, the node which detects the failure sends a report to the network manager nodes. The manager nodes

then identify the failed node/link and try to reroute channels that were using the failed components. All nodes in the system would then be notified about the failed component and the new routes for some of the channels. It may not be possible to support all existing channels, in which case some channels would have to be torn down. Since this failure notification is critical to system operation, the manager node uses a reliable broadcast to propagate it throughout the system.

Failure of one of the manager nodes has to be handled very carefully, since these maintain primary state information about the rest of system. However, these can be handled using techniques similar to those developed in [CM84]. If the node which failed is the token holder, then our channel establishment procedure ensures that the state information is up-to-date on at least one of the other nodes. Such a node will be selected as the new token holder. In the other case, the token holder is guaranteed to have the state information, so there is no loss.

6.5 Current Status

We have ported the *x*-Kernel to the ENP card, and we have also implemented several protocols on it in C and 68000 assembler language. Among the protocols, **hnet** implements the hexagonal mesh link-level protocol, but it uses Ethernet-encapsulate packets for transmission. The request-reply protocol is based on the **chan** protocol (see [HPAO89]) and implements RPC transport. The application interface has also been implemented and it supports both synchronous and asynchronous delivery of messages.

Table 6.4 shows sample performance measurements for some selected operations. The experiments were performed on an otherwise idle network and the operations were repeated 10,000 times to obtain accurate timings. The timing for creating a kernel process was measured by repeatedly creating a process, which immediately destroyed itself. The reported timing therefore is a sum of process create, context switch, and process destroy. The kernel performance is somewhat slower than the figures reported for the Sun-3 because the ENP card has a slow (10 MHz) 68000 processor. Also, some of the processing power is used for a software DRAM refresh. The round-trip times reported are all for packets with 100 bytes data. The ENP-ENP round trip mainly measures the overheads in the Ethernet driver, and the time for processing a packet. Packet processing involves creation of a process to handle the packet. The AP-ENP local call was a "find address" operation and it measures the time for a simple operation, with a few parameters and very little data movement. The AP to AP round-trip time reveals the overheads of crossing the AP to

	Time (milliseconds)
Kernel process create + destroy	0.34
ENP–ENP hnet round–trip	3.16
ENP–ENP chan round–trip	4.81
AP–ENP local call	0.66
AP–AP hnet round–trip	5.60
NM channel create + destroy	23.7

Table 6.4: Communication subsystem performance.

ENP boundary 4 times, and includes the cost of buffer copying.

Among the RTC–related protocols, we have implemented the Network Manager protocol and have done some initial performance measurements on it. The NM was configured to handle a hexagonal mesh of size 3 which has 19 nodes. We first created a base load of 100 channels and then performed the measurements. The channel create operation had to be repeated a large number of times to obtain accurate timings. We were therefore forced to repeatedly create and destroy a channel, so the figure reported above includes channel creation and destruction. The channel created had 3 links and a total end–to–end delay of 100 milliseconds. The same experiment was also repeated on a Sun–3/60, and we obtained a time of 4.1 milliseconds. This large difference is caused by more than just a difference in clock speed between the Sun–3/60 (25 MHz 68020) and the ENP. The channel establishment algorithm contains some integer multiply and divide operations which have to be performed in software on the ENP's 68000 processor. Although the performance of the NM on the ENP is slow, the NP of HARTS is currently being built with an MIPS R3000 processor, so we expect the NP's performance to surpass the Sun–3/60.

It is possible to estimate the timing for channel creation based on the measured request–reply time, the NM create time, etc. The channel creation time seen by the application includes (1) AP–ENP local call, (2) ENP–NM RPC, (3) NM channel create time, (4) ENP–ENP RPCs for registering the channel, and (5) local channel creation overheads. Since the channel contains 3 links, step (4) above requires 3 RPCs. Using the numbers reported above for the different operations, we obtain an application–level channel creation time of 43.5 milliseconds, excluding local channel creation overheads. We expect that the local channel creation overheads will not be substantial: they are mainly updates to the protocol data structures (and server process creation on the source and destination).

Some of the protocols and functions described here have not been implemented because they require hardware support. For example, the clock synchronization protocol

requires a hardware clock and timestamping mechanism, which is not available on the ENP. Similarly, the broadcast operations in the link protocol are currently implemented using Ethernet broadcasts. We have also not completely implemented the RTCP because the send–receive operations are designed for an environment with multiple transmitters and receivers, each with its own queue. We cannot emulate these operations on the ENP which has only a single transmit queue.

We have deferred the implementation of the kernel modifications outlined in Section 6.4.1. The modifications required are mainly in the low-level machine dependent part of the kernel, and as such, they will not be portable to the new NP which uses a MIPS R3000 processor.

APPENDICES

6.A HARTS Packet Format

A brief description of the fields in the packet format, and their effect on packet processing is given below.

```
Message Flags (1 byte)
     Broadcast
     Clock
     Source Route
     Real-time Channel (RTC)
<Route Specification>  (3 bytes)
Destination (2 bytes)
Source (2 bytes)
Length (2 bytes)
Protocol Number (2 bytes)
Priority (4 bytes)
<Protocol-specific part>
<Data>
Checksum (4 bytes)
```

Flags: The Broadcast and Source Route flags are interpreted by the PRC and control how the destination is specified. Consequently, at most one of these two flags can be set for any packet. The Clock flag is interpreted by the PRC and a time-stamp is appended to the packet on transmission and on reception. The RTC flag is ignored by the PRC, but it is essential for the fast dispatching of packets belonging to real-time channels.

<Route Specification> The contents of this field are dependent upon the type of packet.

Destination / Source: The destination and source addresses for this packet. This address is encoded to include broadcast addresses, with different types of broadcast operations each given a separate address. We can consider the address to be composed of a 3 bit **Address Type** and a 13 bit **Host Address.**

Length: The total length of this packet, including the header and the data. Used by the PRC to determine the end of the packet on the send side.

Protocol Number: This field is used to identify different clients of the link level protocols.

Priority: This field is included in the general packet format because it is common across one or more packet formats. It is used to order the packets that are awaiting transmission on a particular link. It is possible to treat this priority to be the deadline for a packet. However, it should be considered to be a per–hop deadline.

<Protocol-specific part> The protocol specific fields of some higher level protocols are given below.

User Datagram Protocol Header

```
Destination Port  (2 bytes)
Source Port       (2 bytes)
```

FRAG Protocol Header

```
Operation Type  (2 bytes)
Client Protocol Number (2 bytes)
Sequence Number (4 bytes)
Number of fragments    (2 bytes)
Fragment mask   (2 bytes)
```

6.B Host User Interface

The following is the list of procedures available to user processes running on the APs to access the communication facilities provided by the NP.

```
XOBJ xcreateprotl (prio,demux,opendone,closedone,control)
     short prio;
     PFI demux, closedone, control;
     PFS opendone;
```

Creates a protocol object which has an agent process of priority `prio` associated with it. If the `demux` procedure is non-nil, the agent process invokes that routine whenever a message arrives from the ENP. Otherwise, the agent places the message in a receive queue associated with the session. The other procedures are handled similarly.

```
xnop(protl)
     XOBJ protl;
```

A dummy operation used to test the host to ENP interface.

```
APSESSN xopen (protl, llp, parts)
     XOBJ protl;
     XOBJ llp;
     PART *parts;
```

Open a session with the lower level protocol `llp` on the ENP, with parameters specified in `parts`.

```
int  xopenenable (p, llp, parts)
     XOBJ p;
     XOBJ llp;
     PART *parts;
```

Listen for a session connection with the lower level protocol `llp`, with parameters specified in `parts`.

```
int xopendisable (hlp, llp, parts)
     XOBJ hlp;
     XOBJ llp;
     PART *parts;
```

Revokes a previously established "openenable".

```
int xacceptsessn (p, sessn)
     XOBJ p;
     APSESSN  *sessn;
```

Accept a new session associated with the host protocol **p**, which has previously been part of an openenable operation. If a session is available, a pointer to this new session is returned in **sessn**. Otherwise, the process may get blocked if WAIT mode is selected (see xcontrolprotl()).

```
int xpush (sessn, msg, msglen, rmsg, rmsglen)
     APSESSN sessn;
     char *msg;
     int  msglen;
     char *rmsg;
     int  *rmsglen;
```

Send a message of length **msglen** contained in the buffer **msg** on the given session. The process blocks until a signal is received from the lower level protocol. If any reply is obtained, it is returned in the buffer **rmsg**, provided this is non-nil, and the length of the reply message is returned in **rmsglen**.

```
int  xclose (sessn)
     APSESSN sessn;
```

Closes the network session associated with **sessn**. It also cleans up the host session structure, releasing any pending messages or blocked processes.

```
int xcontrolprotl (p, protl, opcode, buf, len)
     XOBJ p; /* host protocol */
     XOBJ protl;
     int  opcode;
     char *buf;
     int  len;
```

Use the host protocol object **p** to send a control operation to the protocol **protl** on the ENP. If **protl** is the same as **p**, the operation is local and is executed immediately on the

host. Local operations include submission of receive buffers, adjustment of timeout values for session wait, etc., and their specification is given below. Note that control operations are protocol specific, so remote control operations are determined by the type of protocol (`protl`).

SUBMIT_BUFFER `buf` is the buffer pointer, and `len` is the buffer length of the receive buffer to be submitted.

ACCEPT_WAIT_OFF makes the xacceptsessn() call non-blocking.

ACCEPT_TIMEOUT makes xacceptsessn() blocking, with a timeout value given by `buf`. If the timeout value is 0, an indefinite WAIT is specified.

```
int xcontrolsessn (sessn, localf, opcode, buf, len)
     APSESSN sessn;
     int  localf;
     int  opcode;
     char *buf;
     int  len;
```

Performs local or remote control operations related to the host session, depending upon the value of `localf`. A description of the local control operations is given below.

RECV_WAIT_OFF makes the xrecv() call non-blocking.

RECV_TIMEOUT makes xrecv() blocking. Specifies a timeout `buf` for the blocking, where 0 implies an indefinite wait.

```
XOBJ xgetprotlbyname (p, name)
     XOBJ p;
     char *name;
```

Use the host protocol object `p` to execute a REQ_FINDPROTL command on the ENP to find the protocol corresponding to `name`.

```
xrecv (sessn, buf, len)
     APSESSN sessn;
     char **buf; /* in-out */
     int  *len; /* in-out */
```

Synchronous receive operation. On input, **buf** can point to a buffer of length **len** to be submitted to the pool of receive buffers for the given protocol. If a receive message is available, a pointer to the receive buffer is returned in **buf**, and the length of the message is returned in **len**. If no message is available, the process is blocked awaiting message arrival. The blocked process can be awakened by a timeout, if one is specified (see xcontrolsessn()).

CHAPTER 7

DISCUSSION AND FUTURE WORK

In this chapter we recapitulate the contributions of this dissertation, and explore possible extensions and future directions for the work presented here.

7.1 Research Contributions

We recognize that communication is an important problem in distributed real-time systems. It is necessary to support time-constrained communication, especially between processes that have deadlines associated with their execution. One of the issues involved in supporting time-constrained communication is that we need a mechanism to specify timing constraints. For this purpose, we have developed a framework which allows the specification of the communication requirements of real-time applications. This is based on the abstraction of a real-time channel, which is a unidirectional connection between two ports.

Our attention has been focused mainly on distributed systems with partially connected point–to–point interconnection networks. In this environment, we have identified and solved several problems related to the design of real-time channels. In order to guarantee the delivery time of messages, we have developed algorithms for computing the worst-case delay for messages, and for scheduling these messages. This computation procedure, which is based on priority scheduling, has certain desirable optimality properties. To prevent message loss caused by a lack of buffer space, we developed a mechanism for buffer allocation and flow control suitable for real-time channels.

In a point–to–point interconnection network where there are multiple paths between nodes, route selection is an important problem. Although this problem has been researched extensively for packet switching networks, results for networks with virtual cut-through switching have been lacking. We formulated the problem of routing traffic in a network with virtual cut-through switching as an optimization problem, where the cost of

a route depends upon the links that it uses. We were especially interested in finding routes for traffic on connections like real-time channels, so we imposed the restriction that the traffic on a connection always used the same (static) route. We then selected a link cost function which tries to maximize the probability of establishing virtual cut-through routes in the network using analysis based on a queueing model for the network.

We have shown that the optimization problem is $\mathcal{NP}$-Hard, by proving the associated decision problem to be $\mathcal{NP}$-Complete. Consequently, we developed a heuristic algorithm for the global optimization problem. However, for the special case in which a new route is to be selected without disturbing the routes that have already been established, we have devised a polynomial time optimal algorithm. This case is of great significance because it matches the conditions for real-time channel establishment. Using simulation, we showed that the route selection algorithms gave significant improvements in network performance. The extent of the performance improvement depends upon the topology of the network and the distribution of the sources and destinations of messages.

Reliable broadcasting was the next subject of the dissertation. We first developed a broadcast primitive, RELAY, which is applicable to point–to–point interconnection networks. This primitive has been designed such that it can be easily incorporated with virtual cut-through switching, and it is especially suited for broadcasting in mesh-connected multicomputers. We showed how this RELAY primitive can be easily incorporated into the routing scheme for HARTS, which is the experimental real-time system being developed in the Real-time Computing Laboratory. Using this primitive, it is possible to broadcast a message from any node with only two message transmissions in the best case.

We have also developed a family of reliable broadcast algorithms based on the broadcast primitive. These algorithms, which deliver k copies of a message through node-disjoint paths to each node in the hexagonal mesh, also make efficient use of virtual cut-through. They are especially useful in real-time systems, because they can deliver messages in the presence of undiagnosed faults. We also showed how similar algorithms can be developed for wrapped rectangular meshes. An important application of the reliable broadcast algorithms is in clock synchronization, where the broadcast is used to distribute clock values.

In addition to developing these algorithms for channel establishment, routing, and broadcasting, we have also pursued their implementation in a prototype. We have designed and partially implemented a communication subsystem for HARTS which incorporates support for real-time communication. The communication subsystem, which is targeted for the HARTS network processor (NP), has been implemented, and is currently being tested, on

an Ethernet processor. We have implemented those the parts of the subsystem that are portable to the HARTS NP. Our prototype demonstrates the use of a communication co-processor in providing a high-level abstraction to user processes, thereby offloading protocol processing overheads from the main application processors.

7.2 Future Directions

The Ethernet processor lacks certain hardware features that are essential to some parts of the communication subsystem. For example, the real-time channel transmit–receive module is designed for an environment with multiple transmitters and receivers, each with its own queue. We cannot emulate its functions on an Ethernet processor which has only a single transmit queue. We will implement these parts of the communication subsystem on the HARTS network processor when it is available.

The completion of the implementation will allow us to conduct several interesting experiments. For example, we will be able to measure the performance of the reliable broadcast algorithms using representative workloads. We will also be able to measure the overheads of our clock synchronization algorithm and compare it against other proposed algorithms, such as [OS91]. Moreover, we can experiment with the policies used for scheduling messages to assess their effects on the best-effort class of messages.

This work has revealed several promising research issues that are worth further investigation. The concept of a real-time channel, which is a one–one communication abstraction, can be extended to include one–many connections. This extension, which we term a group channel, is very useful for supporting applications which use replication to achieve fault-tolerance. For example, a group channel can be used to disburse inputs from a sensor station to the replicas of the task which will process the data. The group channel also opens up several issues related to multicast routing, which are possible extensions of our work on route selection.

We have addressed the issue of reconfiguration briefly in this dissertation, but this is an important research topic in its own right. In the present scheme, when a failure occurs there is some message loss before a channel can be reconfigured. It is possible that this interruption in service may not be acceptable for some applications. One possible solution to this problem is to set up standby channels which will facilitate quick reconfiguration. However, we incur the cost of reserving resources for standby channels which could otherwise have been used to allocate other channels. In fact, any solution to this problem must deal with a tradeoff between the reconfiguration time and the resource requirement.

BIBLIOGRAPHY

[ABC+89] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of the Third Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 205–216. ACM, April 1989.

[AF88] D. P. Anderson and D. Ferrari. The DASH project: An overview. Technical Report 84/405, UCB Computer Science Division, EECS, Berkeley, CA, February 1988.

[AHS90] D P. Anderson, R. G. Herrtwich, and C. Schaefer. SRP: A resource reservation protocol for guaranteed performance communication in the internet. Technical Report TR-90-006, International Computer Science Institute, Berkeley, February 1990.

[And88] D. P. Anderson. A software architecture for network communication. In *Proc. 8th International Conference on Distributed Computing Systems*, pages 376–383, June 1988.

[ARS91a] K. Arvind, K. Ramamritham, and J. A. Stankovic. A local area network architecture for communication in distributed real-time systems. *Journal of Real-Time Systems*, 3(2), May 1991.

[ARS91b] K. Arvind, K. Ramamritham, and J. A. Stankovic. Window MAC protocols for real-time communication services. COINS Technical Report 90-127, University of Massachusetts at Amherst, January 1991.

[ATW+90] D. P. Anderson, S. Y. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for continuous media in the DASH system. In *Proc. 10th International Conference on Distributed Computing Systems*, pages 54–61. IEEE, May 1990.

[BJ87] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BS87] R. P. Bianchini and J. P. Shen. Interprocessor traffic scheduling algorithm for multiple-processor networks. *IEEE Transactions on Computers*, C-36(4):396–409, April 1987.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proc. 15th International Conference on Fault Tolerant Computing Systems*, pages 200–206, June 1985.

[CG88] I. Cidon and I. S. Gopal. PARIS: An approach to integrated high-speed private networks. *Intl. Journal of Digital and Analog Cabled Systems*, 1(2):77–86, April 1988.

[CG89] C.-T. Chou and I. S. Gopal. Linear broadcast routing. *Journal of Algorithms*, 10:491–517, December 1989.

[CGGS88] I. Cidon, I. Gopal, G. Grover, and M. Sidi. Real-time packet switching: A performance analysis. *IEEE Journal on Selected Areas in Communications*, 6(9):1576–1586, December 1988.

[CM84] J.-M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Cru87] R. L. Cruz. *A Calculus for Network Delay and a Note on Topologies of Interconnection Networks.* PhD thesis, University of Illinois at Urbana-Champaign, July 1987. available as technical report UILU–ENG–87–2246.

[CSK90] M.-S. Chen, K. G. Shin, and D. D. Kandlur. Addressing, routing, and broadcasting in hexagonal mesh multiprocessors. *IEEE Transactions on Computers*, C-39(1):10–18, January 1990.

[CSRZ90] E. C. Cooper, P. A. Steenkiste, R. D. Ransom, and B. D. Zill. Protocol implementation on the Nectar communication processor. In *Proceedings of the SIGCOMM Symposium*, pages 135–144. ACM, September 1990.

[CY88] D. E. Comer and R. Yavatkar. FLOWS: Performance guarantees in best effort delivery systems. Technical Report CSD-TR-791, Computer Science Department, Purdue University, West Lafayette, IN 47907, July 1988.

[Der74] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings IFIP Congress*, pages 807–813, 1974.

[DRS89] J. W. Dolter, P. Ramanathan, and K. G. Shin. A microprogrammable VLSI routing controller for HARTS. In *Proc. IEEE Int'l. Conf. on Computer Design: VLSI in Computers*, pages 160–163. IEEE, October 1989.

[DRS91] J. W. Dolter, P. Ramanathan, and K. G. Shin. Performance analysis of message passing in HARTS: A hexagonal mesh multicomputer. *IEEE Transactions on Computers*, C–40(6):669–680, June 1991.

[DS86] W. J. Dally and C. L. Seitz. The torus routing chip. *J. Distributed Systems*, 1(3):187–196, 1986.

[DS87] W. J. Dally and P. Song. Design of a self-timed VLSI multicomputer communication controller. In *Proc. IEEE Int'l. Conf. Computer Design: VLSI in Computers*, pages 230–234, 1987.

[EIS76] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal of Computing*, 5(4):691–703, December 1976.

[Fer89] D. Ferrari. Guaranteeing performance for real-time communication in wide-area networks. Technical Report UCB/CSD 89/485, UCB Computer Science Division, EECS, Berkeley, CA, January 1989.

[Fra89] P. Fraigniaud. Asymptotically optimal broadcast and total-exchange algorithms in faulty hypercube multicomputers. Technical Report 89-05, Ecole Normale Superieure de Lyon, 46, Allee d'Italie, 69364 Lyon Cedex 07, France, May 1989.

[FV90] D. Ferrari and D. C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, SAC-8(3):368–379, April 1990.

[G+84] J. Goldberg *et al.* Development and analysis of SIFT. NASA contractor report 172146, NASA Langley Research Center, February 1984.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[Gol90a] S. J. Golestani. Congestion-free transmission of real-time traffic in packet networks. In *Proc. INFOCOM*, pages 527–536. IEEE, June 1990.

[Gol90b] S. J. Golestani. A stop-and-go queueing framework for congestion management. In *Proc. SIGCOMM Symposium*, pages 8–18. ACM, September 1990.

[HP91] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):1–13, January 1991.

[HPAO89] N. C. Hutchinson, L. L. Peterson, M. B. Abbott, and S. O'Malley. RPC in the x-Kernel: Evaluating new design techniques. In *Proc. 12th Symp. on Operating Systems Principles*, pages 91–101. ACM, December 1989.

[IM86] M. Ilyas and H. T. Mouftah. Towards performance improvement of cut-through switching in computer networks. *Performance Evaluation*, 6:125–133, July 1986.

[Jac57] J. R. Jackson. Networks of waiting lines. *Operations Research*, 5(4):518–521, August 1957.

[JH89] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, C-38(9):1249–1268, September 1989.

[Kan] D. D. Kandlur. The *x*-kernel on the ENP-10 Ethernet processor. RTCL working document, December 1990.

[KC88] H. Kanakia and D. R. Cheriton. The VMP network adapter board (NAB): High-performance network communications for multiprocessors. In *Proceedings of the SIGCOMM Symposium*, pages 175–187. ACM, August 1988.

[KK79] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[KKS89] D. D. Kandlur, D. L. Kiskis, and K. G. Shin. HARTOS: a distributed real-time operating system. *ACM SIGOPS Operating Systems Review*, 23(3):72–89, July 1989.

[Kle64] L. Kleinrock. *Communication Nets: Stochastic Message Flow and Delay.* McGraw-Hill, New York, 1964.

[KS90a] D. D. Kandlur and K. G. Shin. Traffic routing for networks with virtual cut-through capability. In *Proc. 10th International Conference on Distributed Computing Systems*, pages 398–405. IEEE, May 1990.

[KS90b] D. L. Kiskis and K. G. Shin. A synthetic workload for real-time systems. In *Proc. Seventh Workshop on Real-Time Operating Systems and Software*, pages 77–81. IEEE, May 1990.

[KSY84] J. F. Kurose, M. Schwartz, and Y. Yemini. Multiple-access protocols and time-constrained communication. *ACM Computing Surveys*, 16(1):43–70, March 1984.

[KSY88] J. F. Kurose, M. Schwartz, and Y. Yemini. Controlling window protocols for time-constrained communication in multiple access networks. *IEEE Trans. Communications*, 36(1):41–49, January 1988.

[KWFT88] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, C-37(4):398–405, April 1988.

[Lal87] J. H. Lala. AIPS tutorial. Technical report, The Charles Stark Draper Laboratory, Inc., January 1987.

[LHA91] J. H. Lala, R. E. Harper, and L. S. Alger. A design approach for ultrareliable real-time systems. *IEEE Computer*, 24(5):12–22, May 1991.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[LMS85] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of ACM*, 32(1), January 1985.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. Real-time Systems Symposium*, pages 166–171. IEEE, December 1989.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LW82] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[OS89] A. Olson and K. G. Shin. Message routing in HARTS with faulty components. In *FTCS-19, Digest of Papers*, pages 331–338, 1989.

[OS91] A. Olson and K. G. Shin. Probabilistic clock synchronization in large distributed systems. In *Proceedings of the 11th Intl. Conference on Distributed Computing Systems*, pages 290–297. IEEE, May 1991.

[RKS90] P. Ramanathan, D. D. Kandlur, and K. G. Shin. Hardware-assisted software clock synchronization for homogeneous distributed systems. *IEEE Transactions on Computers*, C-39(4):514–524, April 1990.

[RS88] P. Ramanathan and K. G. Shin. Reliable broadcast in hypercube multicomputers. *IEEE Transactions on Computers*, C-37(12):1654–1657, December 1988.

[RS91] P. Ramanathan and K. G. Shin. A multiple copy approach to delivering messages under deadline constraints. In *Proceedings of the Twenty-first International Symposium on Fault-tolerant Computing*, pages 300–307. IEEE, June 1991.

[Sch84] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[Sei85] C. L. Seitz. The cosmic cube. *Communications of ACM*, 28(1):22–33, January 1985.

[Shi91] K. G. Shin. HARTS: A distributed real-time architecture. *IEEE Computer*, 24(5):25–35, May 1991.

[SM89] J. K. Strosnider and T. E. Marchok. Responsive, deterministic IEEE 802.5 token ring scheduling. *Journal of Real-Time Systems*, 1(2):133–158, September 1989.

[Sof86] Software Components Group, Inc., Santa Clara, California. *pSOS-68K Real-time Operating System Kernel User's Guide*, March 1986.

[SR88] J. A. Stankovic and K. Ramamritham. *Tutorial: Hard Real-Time Systems.* IEEE Computer Society Press, 1988.

[SS80] M. Schwartz and T. E. Stern. Routing techniques used in computer communication networks. *IEEE Transactions on Communications*, 28(4):539–552, April 1980.

[Ste86] K. S. Stevens. The communication framework for a distributed ensemble architecture. AI Technical Report 47, Schlumberger Research Laboratory, February 1986.

[Str88] J. K. Strosnider. *Highly Responsive Real-Time Token Rings.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, August 1988.

[YM88] C. L. Yang and G. M. Masson. A distributed algorithm for fault diagnosis in systems with soft failures. *IEEE Transactions on Computers*, C-37(11):1476–1480, November 1988.

[ZR87] W. Zhao and K. Ramamritham. Virtual time CSMA protocols for hard real-time communication. *IEEE Transactions on Software Engineering*, 13(8):938–952, August 1987.

[ZSR88] W. Zhao, J. A. Stankovic, and K. Ramamritham. A multi-access window protocol for transmission of time constrained messages. In *Proc. 8th International Conference on Distributed Computing Systems*, pages 384–392, June 1988.