Design of a Communication Subsystem for HARTS

Dilip D. Kandlur

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598
kandlur@watson.ibm.com

Kang G. Shin

Real-Time Computing Laboratory
Dept. of Elec. Engr. and Comp. Sci.
The University of Michigan
Ann Arbor, Michigan 48109–2122.
kgshin@eecs.umich.edu

ABSTRACT

In a distributed real-time system, large or unpredictable delays in the delivery of messages can adversely affect the execution of tasks dependent on these messages. Consequently, it is necessary to associate deadlines with messages between cooperating processes. Therefore, the operating system of a distributed real-time system has to support messages with time constraints, and provide predictable inter-process communication. This paper describes the design of a communication subsystem for HARTS, which is a distributed real-time system being developed at the Real-Time Computing Laboratory, the University of Michigan. This subsystem supports a scheme for predictable inter-process communication in which the system provides *a priori* guarantees for the timely delivery of messages. It also provides other services like clock synchronization and reliable broadcasting, services that are required to support critical real-time applications.

Index Terms — Real-time systems, communications, guaranteed delay, operating systems.

The work reported here was supported in part by an IBM Graduate Fellowship and by the Office of Naval Research under Contract N00014–85–K–0122. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

1 Introduction

A commonly-used definition states that a real-time system 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 [38]. 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, we shall focus on real-time control systems such as aircraft, spacecraft, nuclear power plants, and drive–by–wire automobiles. 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.

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 [16] and MAFT [26] 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.

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 [29, 30]. 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 [36, 10]. 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, thus resulting in a higher total throughput than that of shared medium networks like buses and rings.

Background and Objectives

Communication between real-time 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 goal is to develop a scheme which gives *a priori* guarantees on message delivery for inter-node communication with delivery-time constraints.

Also, 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. Broadcasting is another important operation used in many algorithms such as clock synchronization, distributed diagnosis, and resource lookup. In these algorithms, 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. Therefore, it is necessary to provide support for efficient broadcasting, and for reliable broadcasting in the presence of unknown (or even Byzantine) faults.

In earlier work we have developed solutions to some of these problems, for example, clock synchronization [34] and reliable broadcasting [24]. Typically, 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 [34], we developed a clock 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. For broadcasting, we developed a broadcast primitive which is applicable to interconnection networks with virtual cut-through switching [24]. The primitive is well-suited for broadcasting in mesh-connected multicomputers, where a simple broadcast can be achieved using only two packet transmissions in the best case. We also developed a family of reliable 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 a hexagonal mesh.

In this paper, our aim is to integrate and implement these solutions to provide communication services for applications. Our approach is to develop a communication subsystem which supports real-time communication and provides a global time-base for the system. We use HARTS [37], which is a distributed real-time system being developed at the Real-time Computing Laboratory, The University of Michigan, as the vehicle for our development. The system is comprised of several multiprocessor nodes connected by a point–to–point interconnection network with a hexagonal mesh topology. A HARTS node (see Fig. 1) 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.

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, which in this case is the NP, to provide these services. 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. Moreover, the added cost of a dedicated NP can be offset by its use for other system functions like 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 [19] implements some link and network level protocols in the hardware. It is specifically designed for the VMTP protocol, and it provides a

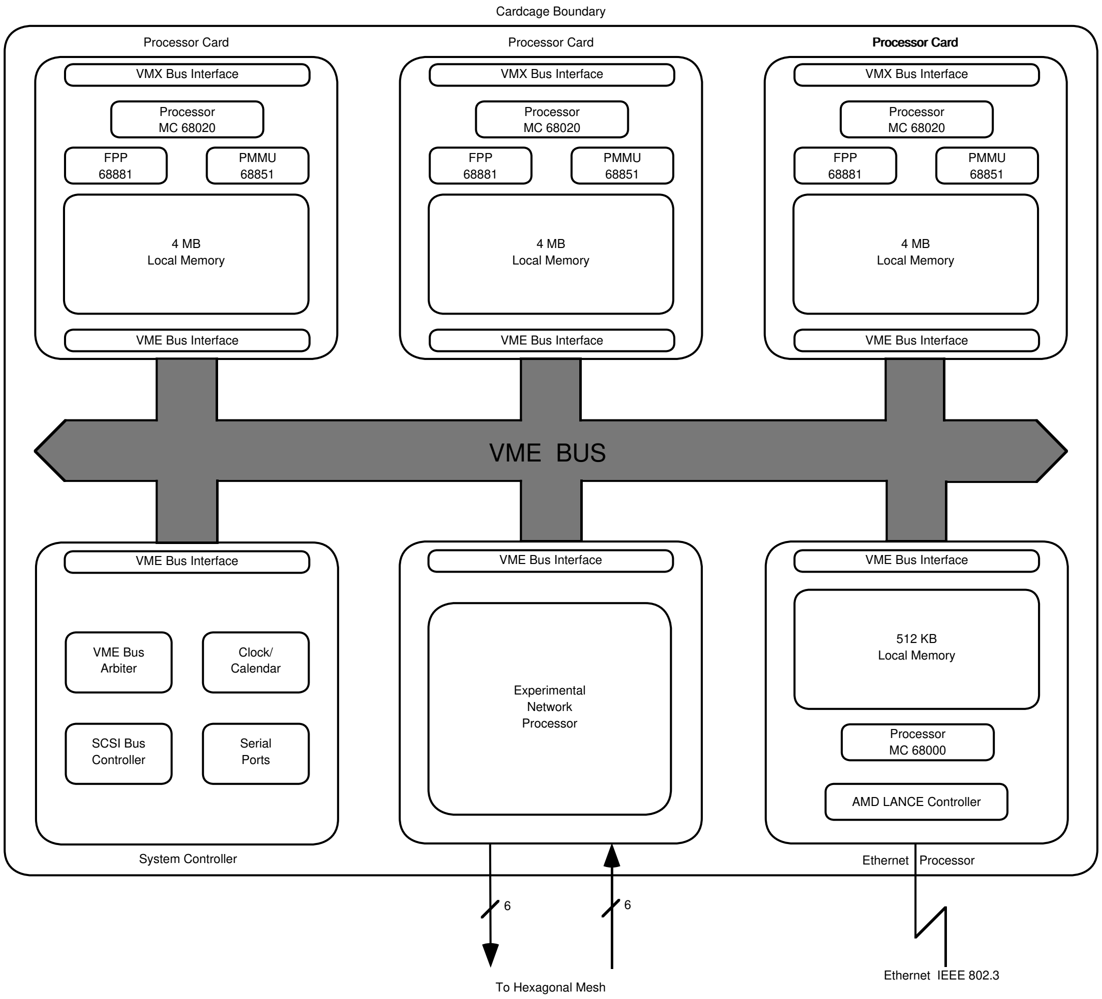


Figure 1: Block diagram of a HARTS node.

"device" interface to the application processors which are responsible for executing higher level protocols. The Nectar system [5] uses a communications adapter board (CAB) which features a general-purpose SPARC processor. The CAB software organization [11] is similar to our design and provides a high-level interface to the host processors. However, these systems do not deal with the problem of real-time communication.

This paper is organized as follows. In Section 2 we describe the communication model that we use for time-constrained communication, which is based on an abstraction called a *real-time channel*. Section 3 outlines the algorithms used to support real-time channels. Section 4 presents an overview of the communication subsystem and its environment, while Section 5 describes some of the services that it provides, such as clock synchronization. In Section 6 we present our design for providing the real-time channel service in HARTS. We discuss the current state of the implementation in Section 7. The paper concludes with Section 8.

2 Communication Model

Our objective 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. This problem 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. Many of these efforts have been directed towards designing medium access protocols for multiple-access networks which consider time constraints on messages. For example, in the case of CSMA/CD networks, the proposed protocols include the virtual-time based methods [41] and the window-based access methods [28, 42]. The survey paper by Kurose *et al.* [27] 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 provide guarantees about their delivery times. For example, Strosnider and Marchok [40] 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 [6, 7] 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 fault-tolerance 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 [15, 2, 3] dealing with continuous-media communication in wide-area networks is related to the work reported here. Our approach to providing guaranteed real-time communication is similar to the one presented in [15]. However, our method for computing the guarantees is more general and we can show the delay bound test of [15] as a special case of our method. Moreover, we present an integrated solution to the problem of buffer reservation and flow-control.

2.1 Message Types

In a real-time system, there are several classes of messages with different requirements. As was done in [39], we classify message traffic into three classes:

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

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

Real-time messages, which have time-constraints on their delivery time, form the second class. These include both periodic and aperiodic messages. For periodic messages, the periodicity can be precisely specified and the inter-arrival time is approximately constant. An important subclass of this are the *Clock* messages which are exchanged between nodes as part of a clock synchronization algorithm. The arrival pattern for aperiodic messages can be specified using a pessimistic estimate for the minimum inter-arrival time. We assume 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 real-time messages. The multiple-copy approach (using multiple disjoint paths) 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-constraints have 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 use a communication abstraction which is oriented toward real-time messages.

2.2 Real-time Channels

Communication between user-level entities 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 real-time applications which require guaranteed delivery time for communication [15]. 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 a 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 CPU time for message processing. 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 hosts, 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 cannot provide a guarantee.

Message Generation Model: Since a significant portion of the traffic which requires guarantees will be periodic, 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 [12]. This model has also been adopted by Anderson and others [1, 4] for continuous media applications, and some of the terminology given below is from [1]. 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-generation time between messages. The burst parameter B_{max} puts a bound on the allowed short-term variation in the message generation, 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-generation time and the average rate of generation.

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, and they are adequate to 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

These attributes are sufficient to describe the requirements of the three message types, but 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 Section 6. For non real-time messages the message generation parameters are not necessary since resources are not reserved for this class.

3 Supporting Real-time Channels

There are several interesting issues associated with the design and implementation of real-time channels. Conceptually, there are two phases to be considered: channel establishment and message transmission. The first phase deals with creation of the channel and falls in the domain of network management. Actions taken in the message transmission phase include message scheduling, flow control, buffer management, etc. These two phases are closely related because the computation of the end–to–end delivery time must be based on the scheme used for message scheduling.

3.1 Channel Establishment

The actions taken by the communication subsystem to establish a channel depend upon the type of channel. They include allocation of resources, route selection, verification of timing properties, etc.

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 Alert 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 disjoint paths to the destination, as specified in the reliability field. These paths have to be 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 [35] show how the multiple copy approach can improve the probability of timely delivery for messages in addition to tolerating component failures.

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.

Real-time messages: The channel establishment operation is quite complicated in this case, and its detailed account is given in [25]. It is summarized here for completeness.

In a point–to–point interconnection network, there are two alternatives for routing messages in the network, namely dynamic and static routing. We employ static routing for real-time messages because it is very difficult to make any guarantees on message delivery time for a channel based on dynamic routing, in which a message can use one of several alternate routes through the network. The channel establishment phase is outlined in Fig. 2, and begins with the selection of a route for the channel. The timing property has to be verified by computing the worst-case delay for a message on each link of the selected route. In this computation, we have to ensure that the new channel does not affect the guaranteed delivery times of existing channels. We also have to compute the buffer requirement for this channel, both at the end nodes and at intermediate nodes, and reserve buffer space accordingly. This is necessary in order to prevent buffer overruns and consequent loss of messages.

The worst-case delay for a message at a link depends upon the other channels which use the link and upon the scheduling algorithm. Our channel establishment scheme uses fixed-priority scheduling for computing the worst-case delay, but we use a form of Earliest Due Date (EDD) as the run-time message scheduling algorithm. The EDD algorithm has been shown to be the optimal scheduling algorithm for many different environments, including preemptive scheduling of periodic tasks [13, 32]. However, it is difficult to determine whether a given task

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.
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 delay test.
4. If the channel passes the delay test, divide the user-specified delay among the links on the route based on their worst-case delay for the message. Compute the buffer requirement for this channel based on these allocated delays and compare it with the available buffer space. This is the buffer requirement test.
5. The channel can be established if it passes both the channel delay and the buffer requirement tests.

Figure 2: **Channel Establishment Procedure**

set has a feasible EDD schedule. We therefore use fixed-priority scheduling as a heuristic in determining the feasibility. 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 [25]. 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. 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 than the former. 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 when assigning the priority of each channel as described in [25].

3.2 Run-time 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 i–th message on the channel M_c. All packets belonging to the i–th 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 2, where t_i denotes the generation time of message m_i:

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

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. 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

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

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 generation time at the source node. This definition of the logical arrival time is feasible because the nodes in the system have synchronized clocks. We assume that the maximum skew between clocks on different nodes is small compared to the message delivery delays, and skews of this order can be realized (as in [34]).

The logical arrival time that is assigned to messages is used by the message scheduler, which is based on a variation of the multiclass 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 at a node maintains the following 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 the order of increasing deadlines.

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

1. Find $\ell_c(m_i)$, the logical arrival time of the message to which this packet belongs.

2. Set $\ell t(\text{packet}) := \ell_c(m_i)$.

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: **Processing on Packet Arrival**

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, but which can have deadlines associated with them. For example, packets belonging to messages which request "best-effort" type service would fall in this category. 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 service stations. These packets are stored in 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 Fig. 3, 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 first checks whether any real-time packets in Queue 3 have become current, and transfers all 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 a bound on the separation between the logical arrival time of a message and the real time, and it controls the buffer requirement of a channel. For any link, the horizon is determined by the amount of buffer memory available at the downstream node for transient packets (recall that links are uni-directional), and is re-computed whenever a new channel is established with that link. This computed horizon is passed back to the upstream node and that node is responsible for flow-control on the link, as described above. The principal advantage of using a per-link horizon is that it achieves a rate-based flow-control without using timers for each channel. It is conceivable to use a different horizon for each channel, but the run-time scheduler would then be forced to examine all packets in Queue 3 individually to check their eligibility for transmission.

Packets in Queue 3 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 an appropriate future instant.

4 The Communication Subsystem

The HARTS communication subsystem has been designed to provide communication related services to applications and mechanisms for reconfiguration in the case of network component failures. This subsystem offers services such as the real-time channel service and maintains a global time-base for the system using a clock synchronization algorithm. The global time-base has benefits not only for application tasks, but also for several system functions like message scheduling. The subsystem also incorporates other functions like reliable broadcasting. It is designed to be resident on the Network Processor (NP) and it is interfaced to the real-time kernel which runs on the Application Processors (APs).

Table 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

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

Table 1: Communication subsystem structure.

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 supported at this level. It will be seen that this mechanism can benefit greatly from hardware assistance at the physical level. 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. As mentioned earlier, the real-time channel service is our method for providing time-constrained communication. The network management functions include the establishment of real-time channels and handling of link/node failures.

The HARTS NP is currently under development, and a block diagram is shown in Figure 4. 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 [14] and 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 here has, however, been implemented on the ENP-10 Ethernet processor, because the construction of NP has not been completed. 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; in the place of the IMU it has a Motorola 68000 processor. Thus, in the

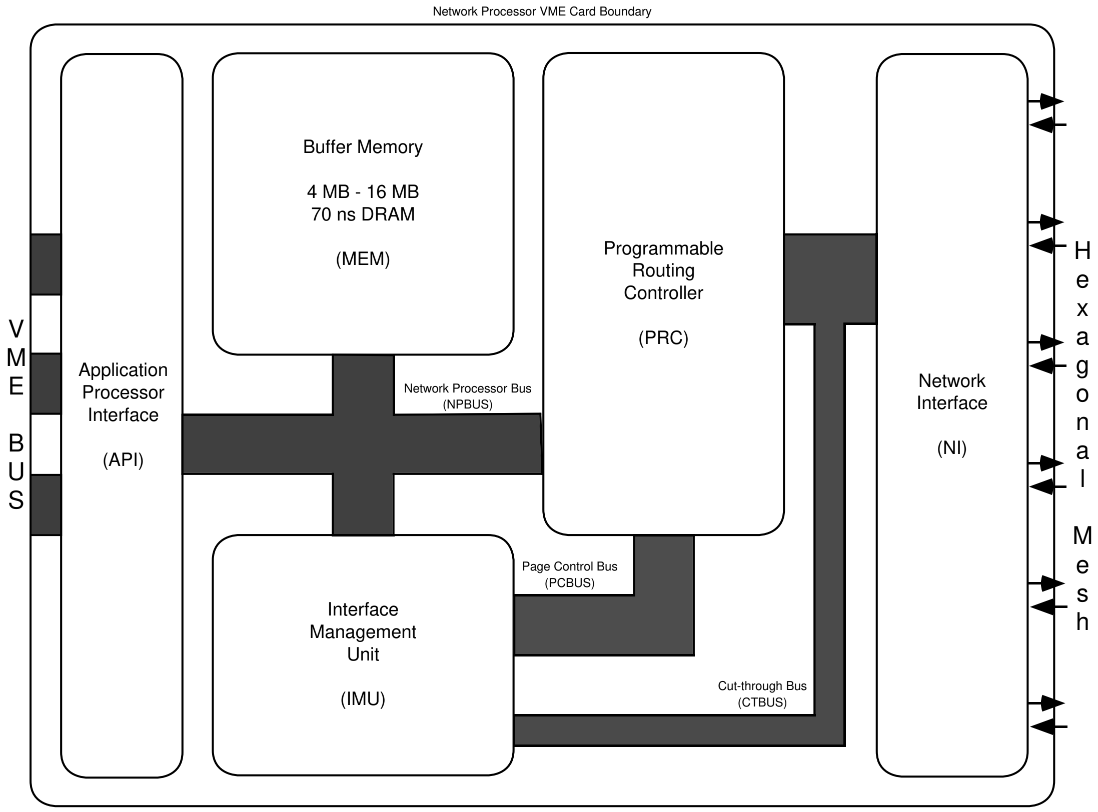


Figure 4: Network processor architecture.

discussion that follows, we take the liberty of using the terms "NP" and "ENP" interchangeably.

The NP kernel

We have employed a derivative of the x–Kernel [17] 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 supports three types of communication objects: *protocols*, *sessions*, and *messages*. Protocols are static, passive objects and each protocol object corresponds to 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 kernel also comes with utilities to configure and

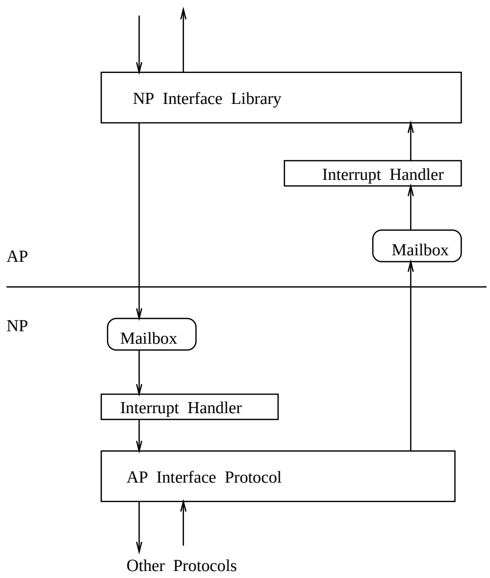


Figure 5: The host interface.

test different protocol stacks. However, several modifications are necessary to make it support the real-time communication service which are described in Section 6.1.2.

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. The changes required for the ENP implementation can be attributed to (a) differences between the 68020 and 68000 architectures and instruction sets 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. For example, the ENP–10 does not have any memory mapping hardware, nor does it make any distinction between user and supervisor state for memory access. Hence, the process management routines have been modified to remove all virtual address space support. Details of this port can be found in [20].

The application programmer interface to the communication subsystem has been designed to provide both synchronous and asynchronous forms of message delivery. The interface allows uniform access to any protocol on the NP, and is implemented as a layer which is split across the NP and the APs. Signaling between the NP and the APs is accomplished using mailboxes in memory, and inter-processor interrupts. Fig. 5 shows a block diagram of the interface, further details can be found in [21].

5 Communication Services

Communication services are provided in the form of protocols running on the x–Kernel. Fig. 6 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.

Fig. 6 also shows several protocol modules related to the real-time channel (RTC) service which will be described later in Section 6. 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 indicates that the Network Manager, which is a centralized service that handles RTC establishment and maintenance, is not present on all nodes in the system.

Before describing these protocols, we will first present the packet format that we have developed for the HARTS. This 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, source and destination addresses, priority, and other fields. 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 the 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.

5.1 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 the LLP 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.

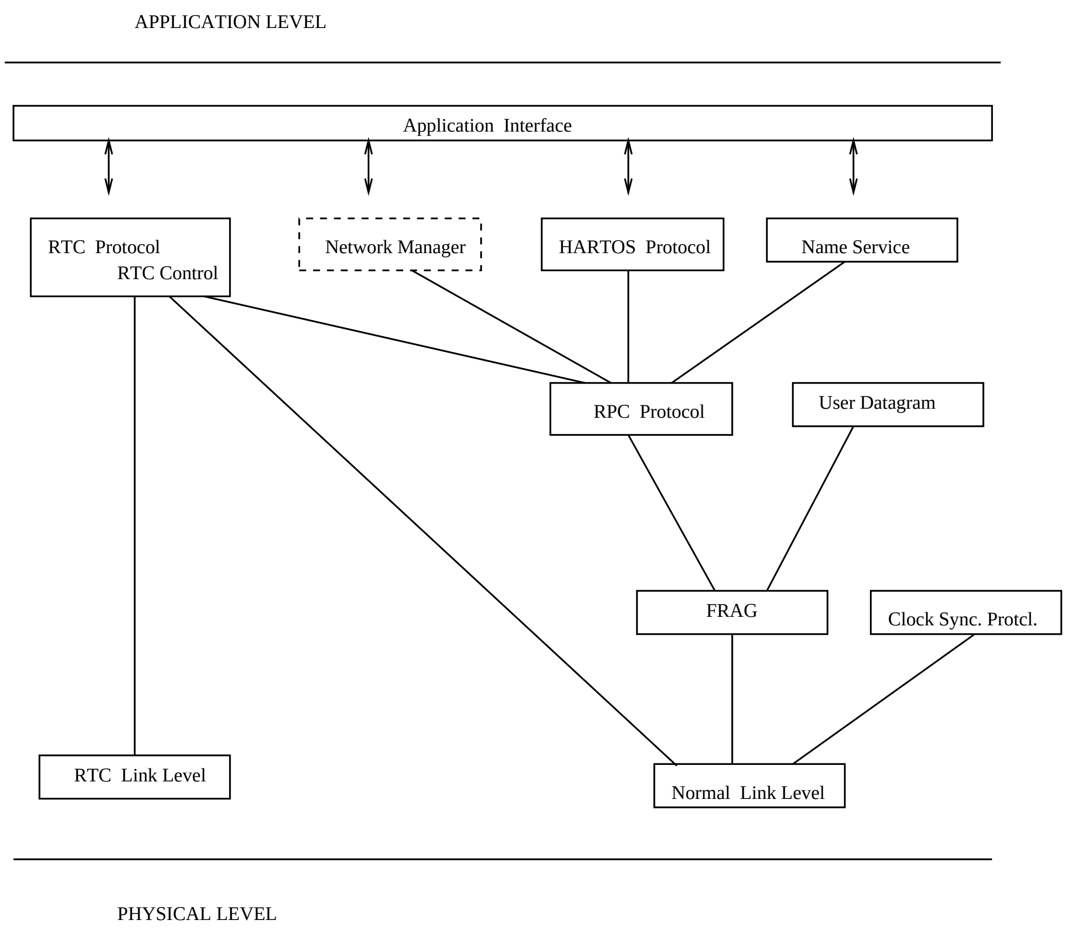


Figure 6: Simplified view of the communication subsystem.

Different types of broadcast operations are supported: from simple broadcasts to 6-reliable broadcasts, and the type of broadcast is determined by the destination field in the packet. Broadcast packets also carry some additional information in the route specification field, *hop count*, *tag* and *step*, which is required to implement the broadcast algorithms of [24]. 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 [24]. 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.

5.2 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 uses the clock synchronization scheme presented in [34] since it allows moderately tight synchronization and does not require a separate clock network. This scheme relies on a hardware timestamping mechanism to disseminate its clock value to other nodes [34] and uses the Interactive Convergence Algorithm[31] on the clock values that it receives from other nodes. The network interface must affix a transmit timestamp to a clock packet just before its transmission and it must append 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. A clock packet contains the following information: the node at which the broadcast was initiated (the initiator), the node from which the message was most recently relayed (the immediate sender), and five words W_1, W_2, ..., W_5 shown below.

- W_1: Clock time of the initiator when this message was transmitted.
- W_2: Clock time of the immediate sender when it received this message.
- W_3: Clock time of the immediate sender when it relayed this message.
- W_4: Accumulated transit delay not counting the delay at the immediate sender.
- W_5: The time on the local clock when this message was received.

The description in [34] assumes that the clock used for timestamping is the same as the node clock. In practice, the timing constraints imposed by the timestamping mechanism on the send–receive hardware are such that it is necessary to use more than one clock. The PRC has to access the clock value immediately after the end of a packet is detected, and it may not be able to arbitrate for the NP bus and access the clock if it is off-chip. Hence, it uses an on-chip clock counter for the timestamping, which can also be read and written by the IMU. However, due to area limitations on the PRC chip this clock counter does not provide any timer functions. The NP therefore contains two clocks: a clock/timer chip which is capable of generating timer interrupts for the IMU, and the clock counter inside the PRC. These clocks are driven by the same crystal so they operate at the (approximately) same rate. The CP uses the PRC clock only for interval measurements, so its absolute value need not be identical to the value of the external clock.

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 5-reliable broadcast allows us to tolerate up to two Byzantine faults, this being the maximum number for the six-connected hexagonal mesh. 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.

5.3 Other Protocols

The RPC protocol implements a request–reply operation with *at most once* semantics, using the technique of Birrell and Nelson [8]. It 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. The fragment protocol (FRAG) 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 and is used to transport both RPC and UDP packets. 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.

We can implement the existing HARTOS calls [22] 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.

6 Real-time Channel Service

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 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.

6.1 Real–time Channel Protocol

This protocol implements the real-time channel communication scheme and handles the different types of messages described in Section 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 called the real-time channel controller (RTCC).

6.1.1 Channel Control

The control part handles requests for channel creation and teardown, and interfaces with the NM using the services of RPC as shown in Fig. 6. 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 RPCs 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.

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 subsection. The session pointer is returned to the requesting task and 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.

6.1.2 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. 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.

6.1.3 Message Transmission

RTCP treats each send request to be a separate message. It handles messages of varying sizes, and in case messages 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.

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.

6.1.4 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.

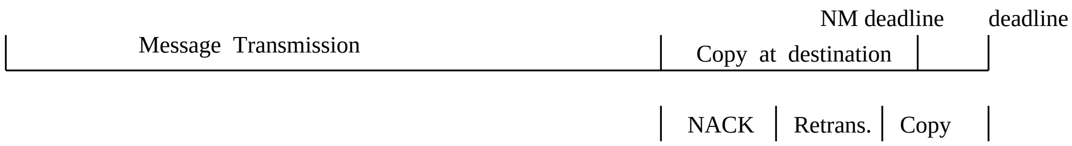
Short messages are handled by transmitting two copies of the message over the same route at two different times, and the destination is responsible for choosing a copy from the packets that it receives. The advantage of this scheme is that it is simple and does not require any acknowledgment from the 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 negative acknowledgment (NACK) from the destination to the source, 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 shorter 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. Note that this use of a shorter deadline is not free of cost, it imposes tighter scheduling constraints and could reduce the number of RTCs that can be accommodated.



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.

6.2 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 establishment scheme outlined in Section 3.1. The NM maintains several structures which are vital to the channel establishment procedure. It maintains a table containing the resource requirements and the assigned routes 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. 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 an incremental routing algorithm, described in [23], to select routes for channels. This algorithm uses the link loads imposed by existing channels as a parameter in the selection of the route for the new channel. 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 presented in [25], 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. The latter 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 (see Section 3.2). 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 of NM: 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 NM on multiple nodes. However, this gives rise to the problem of maintaining consistency amongst the replicated tasks. We have to make sure that that the replicas of the NM coordinate their activities to avoid inconsistencies.

We have developed a scheme for replication in which the NM 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 [9]. One of the three NMs 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.

Component 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 NM nodes. The NM 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 NM node uses a reliable broadcast to propagate it throughout the system.

Failure of one of the NM nodes has to be handled very carefully, since these maintain primary state information about the rest of system. A failure of this type can be handled using techniques similar to those developed in [9]. 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.

7 Current Status

We have ported the x–Kernel to the ENP card, and also implemented several protocols on it in C and 68000 assembler language. Among the protocols, `hnet` implements the hexagonal mesh link-level protocol, and uses Ethernet encapsulation for packet transmission. The request–reply protocol is based on the `chan` protocol (see [18]) and implements RPC transport. The application interface has also been implemented and supports both synchronous and asynchronous delivery of messages.

Table 2 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 [17] 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

	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 2: Communication subsystem performance.

overheads of crossing the AP to ENP boundary 4 times, and includes the cost of buffer copying.

Among the RTC–related protocols, we have implemented the NM 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 IMU of the HARTS NP is a 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 6.1.2. 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.

8 Summary

We have presented the design and implementation of a scheme for time-constrained communication in real-time systems. The implementation is in the form of a communication subsystem for an experimental distributed real-time system, called HARTS. The communication subsystem also offers other important services such as maintenance of a global time-base. This subsystem is targeted for the HARTS Network Processor (NP), but it has been implemented on an Ethernet processor.

The completion of the implementation on the HARTS NP 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 [33]. Moreover, we can experiment with the policies used for scheduling messages to assess their effects on the best-effort class of messages.

Acknowledgments

We would like to thank James Dolter, Stuart Daniel, and Ashish Mehra for the many useful discussions regarding the design of the HARTS NP. We also thank the developers of the x–Kernel for making their sources available.

References

[1] 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*, pp. 54–61. IEEE, May 1990.

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

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

[4] 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.

[5] 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)*, pp. 205–216. ACM, April 1989.

[6] 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*, vol. 3, no. 2, , May 1991.

[7] 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.

[8] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, February 1984.

[9] J.-M. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 251–273, August 1984.

[10] M.-S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, routing, and broadcasting in hexagonal mesh multiprocessors," *IEEE Transactions on Computers*, vol. C-39, no. 1, pp. 10–18, January 1990.

[11] 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*, pp. 135–144. ACM, September 1990.

[12] 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.

[13] M. L. Dertouzos, "Control robotics: The procedural control of physical processes," in *Proceedings IFIP Congress*, pp. 807–813, 1974.

[14] 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*, pp. 160–163, Boston, MA., October 1989, IEEE.

[15] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. SAC-8, no. 3, pp. 368–379, April 1990.

[16] J. Goldberg et al., "Development and analysis of SIFT," NASA contractor report 172146, NASA Langley Research Center, February 1984.

[17] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[18] 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*, pp. 91–101. ACM, December 1989.

[19] H. Kanakia and D. R. Cheriton, "The VMP network adapter board (NAB): High-performance network communications for multiprocessors," in *Proceedings of the SIG-COMM Symposium*, pp. 175–187. ACM, August 1988.

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

[21] D. D. Kandlur, *Networking in Distributed Real-time Systems*, PhD thesis, The University of Michigan, Ann Arbor, Rackham School of Graduate Studies, Ann Arbor, Michigan 48109, August 1991.

[22] D. D. Kandlur, D. L. Kiskis, and K. G. Shin, "HARTOS: a distributed real-time operating system," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 3, pp. 72–89, July 1989.

[23] 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*, pp. 398–405. IEEE, May 1990.

[24] D. D. Kandlur and K. G. Shin, "Reliable broadcast algorithms for HARTS," *ACM Transactions on Computer Systems*, vol. 9, , November 1991.

[25] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," in *Proceedings of the 11th Intl. Conference on Distributed Computing Systems*, pp. 300–307. IEEE, May 1991.

[26] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai, "The MAFT architecture for distributed fault tolerance," *IEEE Transactions on Computers*, vol. C-37, no. 4, pp. 398–405, April 1988.

[27] J. F. Kurose, M. Schwartz, and Y. Yemini, "Multiple-access protocols and time-constrained communication," *ACM Computing Surveys*, vol. 16, no. 1, pp. 43–70, March 1984.

[28] J. F. Kurose, M. Schwartz, and Y. Yemini, "Controlling window protocols for time-constrained communication in multiple access networks," *IEEE Trans. Communications*, vol. 36, no. 1, pp. 41–49, January 1988.

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

[30] J. H. Lala, R. E. Harper, and L. S. Alger, "A design approach for ultrareliable real-time systems," *IEEE Computer*, vol. 24, no. 5, pp. 12–22, May 1991.

[31] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal of ACM*, vol. 32, no. 1, pp. 52–78, January 1985.

[32] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

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

[34] P. Ramanathan, D. D. Kandlur, and K. G. Shin, "Hardware–assisted software clock synchronization for homogeneous distributed systems," *IEEE Transactions on Computers*, vol. C-39, no. 4, pp. 514–524, April 1990.

[35] 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*, pp. 300–307. IEEE, June 1991.

[36] C. L. Seitz, "The cosmic cube," *Communications of ACM*, vol. 28, no. 1, pp. 22–33, January 1985.

[37] K. G. Shin, "HARTS: A distributed real-time architecture," *IEEE Computer*, vol. 24, no. 5, pp. 25–35, May 1991.

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

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

[40] J. K. Strosnider and T. E. Marchok, "Responsive, deterministic IEEE 802.5 token ring scheduling," *Journal of Real-Time Systems*, vol. 1, no. 2, pp. 133–158, September 1989.

[41] W. Zhao and K. Ramamritham, "Virtual time CSMA protocols for hard real-time communication," *IEEE Transactions on Software Engineering*, vol. 13, no. 8, pp. 938–952, August 1987.

[42] 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*, pp. 384–392, June 1988.