HARTS: A Distributed Real-Time Architecture

Kang G. Shin, University of Michigan

The growing importance of real-time computing in numerous applications, such as aerospace and defense systems and industrial automation and control, poses problems for computer architectures, operating systems, fault tolerance, and evaluation tools. The interplay of three major components characterizes real-time systems. First, time is the most precious resource to manage. Tasks must be assigned and scheduled to be completed before their deadlines. Messages must be sent and received in a timely manner between the interacting real-time tasks. Second, reliability is crucial, since failure of a real-time system could cause an economic disaster or the loss of human lives. Third, the environment in which a computer operates is an active component of any real-time system. For example, in a "drive by wire" transportation system, in which important functions such as emission control and braking are automated with computers, it would be meaningless to consider the on-board computers without considering the automobile itself.

Because their multiplicity of processors and internode routes gives them the potential for high performance and high reliability, distributed systems with point-to-point interconnection networks are natural candidate architectures for time-critical applications. This article focuses on the design, implementation, and evaluation of a distributed real-time architecture called HARTS (hexagonal architecture for real-time systems), with emphasis on its support of time-constrained, fault-tolerant communications and I/O requirements. Currently under development at the University of Michigan's Real-Time Computing Laboratory (RTCL), HARTS consists of shared-memory multiprocessor nodes, interconnected via a wrapped hexagonal mesh. This architecture is intended to meet three main requirements of real-time computing: high performance, high reliability, and extensive I/O.*

Consisting of shared-memory multiprocessor nodes interconnected by a wrapped hexagonal mesh, HARTS is designed to meet the special communications and I/O needs of time-critical applications.

High-level architecture

The primary goal of HARTS is the study of low-level architectural issues, such as message buffering, instruction set design, scheduling, and routing, in a setting that gives designers internal access to many system parameters. To meet this goal, my colleagues at RTCL and I used a hybrid system of commercially available processors and custom-designed interfaces. Several processor cards are grouped to form a cluster of application processors (APs). Each cluster serves as a multiprocessor node and is interconnected by custom interfaces to form a distributed system. The presence of both multiprocessor and distributed aspects permits investigating the behavior of real-time tasks under either architectures. In parallel with the hardware

*Although predictability of task execution behavior is essential for any real-time system design to guarantee on-time completion of tasks, it is not an architectural issue but an operating system issue. Real-time systems architects must provide hardware facilities on which one can readily build an operating system that guarantees deadlines.

0018-9162/91/0500-0025$01.00 © 1991 IEEE

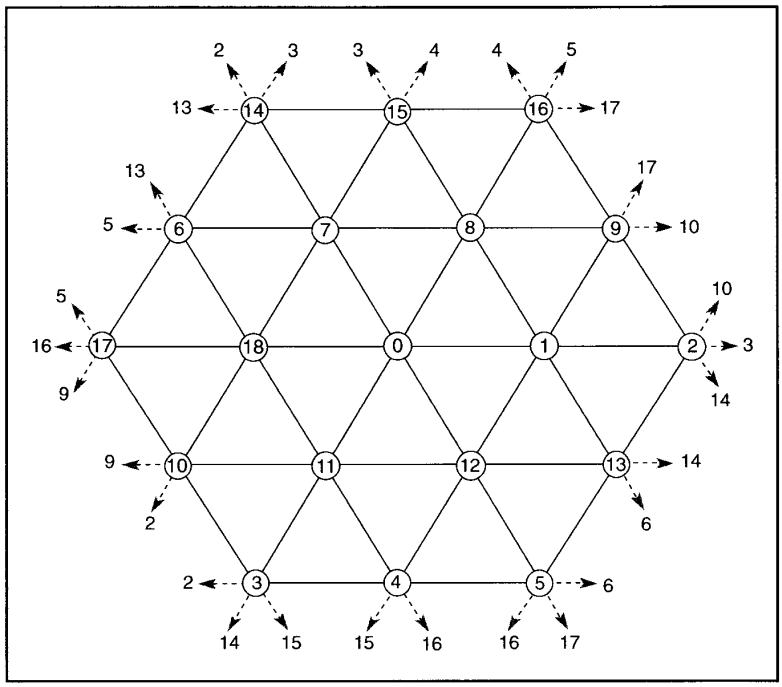

Figure 1. Hexagonal mesh of size 3.

development, our work on a real-time operating system called HARTOS[1] influences the specification, architecture, and implementation of the custom-designed HARTS components.

Node architecture. Each node in the testbed consists of up to three APs, a system controller, a shared memory segment, an Ethernet processor, and the network processor (NP), a custom-designed interface to the interconnection network.

The APs are commercially available VME bus multiprocessing engines based on Motorola's MC68020. Each processor card has four major sections: a CPU core, a VME bus interface, a memory system, and a VMX bus interface. The CPU core consists of a 32-bit 16-MHz MC68020 CPU, an MC68881 floating-point processor, and an MC68851 paged memory management unit. The memory subsystem provides four megabytes of high-performance dual-ported dynamic RAM, with an additional 256 bytes of special mailbox hardware. This mailbox feature facilitates efficient interprocessor communication by allowing a remote processor to write a semaphore that automatically interrupts the local processor.

The Ethernet processor card supports several functions for the nodes, although it is not a permanent HARTS component. First, it provides a secondary means of distributing code and data. This is especially important during the early stages of development of the network interfaces. Second, the high-level protocols that manage reliable packet handling and provide internode communication can be experimentally tested on the Ethernet processor. Third, the Ethernet processor is used to collect experimental data by monitoring the APs and network interfaces with minimal interference.

Interconnection network. A distributed system's interconnection network often connects thousands of homogeneously replicated processor-memory pairs, which are called processing nodes (PNs). All synchronization and communication between PNs for program execution are via message passing. The homogeneity of PNs and the interconnection network is very important because it allows cost and performance benefits from the inexpensive replication of multiprocessor components.[2] It is preferable that each PN in the multiprocessor have fixed connectivity so that standard VLSI chips and communication software can be used. Also, the interconnection network should contain a reasonably high degree of connectivity so that alternative routes can be made available to detour faulty nodes and links. More important, the interconnection network must facilitate efficient routing and broadcasting to achieve high performance in task execution. For structural flexibility, a system must also possess fine scalability, measured in terms of the number of PNs necessary to increase the network's dimension by one.

To meet these requirements, we considered several topologies, including hypercubes, square meshes, 3D tori, hexagonal meshes, and octal meshes. Of these, the hexagonal (H) mesh best meets the requirements of fixed connectivity and planar architecture for easy VLSI and communications implementation, fine scalability, reasonably high fault tolerance, and ease of construction. (Detailed comparisons of the H-mesh to other topologies are given by Stevens[2] and by Chen, Shin, and Kandlur.[3] The robustness of an H-mesh to link and node failures is shown by Olson and Shin.[4]) Hence, we chose a C-wrapped ("C" stands for continuous) H-mesh topology to interconnect HARTS nodes.

H-mesh size (the term "dimension" was used in an earlier article[3]) is defined as the number of nodes on its peripheral edge. One can visualize what is happening in the C-wrapping by first partitioning the nodes of a nonwrapped H-mesh of size e, denoted by H_e, into rows in three different directions. The mesh can be viewed as composed of $2e-1$ horizontal rows (called the d_0 direction), $2e-1$ rows in the 60-degree clockwise direction (called the d_1 direction), or $2e-1$ rows in the 120-degree clockwise direction (called the d_2 direction). In each of these partitions we label from the top the rows R_0 through R_{2e-2}. The C-wrapping is then performed by connecting the last node in R_i to the first node in $R_{[i+e+1]2e-1}$ for each i in each of the three partitions, where $[a]_b$ denotes a mod b. Figure 1 illustrates an example of a C-wrapped H_3 in which the links on the periphery (represented by dashed-line arrows) are connected to the nodes as indicated by their labels.

The C-wrapped H-mesh is isomorphic to the interconnection topology presented by Stevens.[2] However, the formalism just described allows uniform treatment of message routing between all pairs of nodes and does not require any special treatment of the wrap lines, as was necessary in

Stevens' topology when the axial offset was between e and $2(e-1)$.

A C-type wrapping has several salient properties, as shown by Chen, Shin, and Kandlur.[3] First, this wrapping results in a homogeneous network. Consequently, any node can view itself as the center of the mesh (labeled as node 0 in Figure 1). Second, the diameter of an H_e is $e-1$. Third, there is a simple, transparent addressing scheme that uses only one coordinate — instead of three as in Stevens' topology — to uniquely identify any node in an H-mesh. An example of this addressing for an H_4 is shown in Figure 2a, where all edges are omitted for clarity. On the basis of this addressing scheme, one can determine the shortest paths between any two nodes with a $\Theta(1)$ algorithm; that is, the complexity is constant and independent of system size. Note that at each node on a shortest path there are at most two different neighbors of the node to which the shortest path runs. Fourth, with this addressing scheme one can devise a simple routing algorithm that can be efficiently implemented in hardware, as shown by Dolter, Ramanathan, and Shin.[5]

To send a message, the source calculates the shortest paths to the destination and encodes this routing information into three integers denoted by m_0, m_1, and m_2, which represent the number of hops from the source to the destination along the d_0, d_1, and d_2 directions, respectively. Before sending the packet to an appropriate neighbor, intermediate nodes update these values to indicate the remaining hops in each direction to the destination. Hence, $m_0 = m_1 = m_2 = 0$ indicates that the packet has reached its destination.

Suppose node 11 sends a message to node 5 in the H_4 of Figure 2. The original H_4 is given in Figure 2a and $H_4(11)$ — node 11 is placed at the center of the H_4 — is in Figure 2b. From the Chen-Shin-Kandlur routing algorithm, we get $m_0 = 0$, $m_1 = -2$, and $m_2 = -1$. Note that the route from node 11 to node 5 in Figure 2b is isomorphic to that from node 0 to node 31 in Figure 2a. This is not a coincidence but rather a consequence of the homogeneity of H_4.

Applications in various domains require an efficient method for a node to broadcast a message to all the other nodes in an H-mesh. Due to interconnection costs, it is very common to use point-to-point communications for broadcasting. Without loss of generality, one can assume the center node is the broadcast source. The set of nodes that has the same distance from the source node is called a ring. The main idea of this algorithm is to broadcast a message, ring by ring, toward the periphery of an H-mesh. The algorithm consists of two phases. In the first phase, which takes three steps, the message is transmitted to the origin's six nearest neighbors. Note that there are six corner nodes in each ring. In the second phase, which takes $e-1$ steps, the six corner nodes of each ring send the message to two neighboring nodes, while all other nodes propagate the message to the next node in the same direction as the previous transmission. Figure 3 is an example of H_4 broadcasting. The numeric labels denote the communication step numbers.

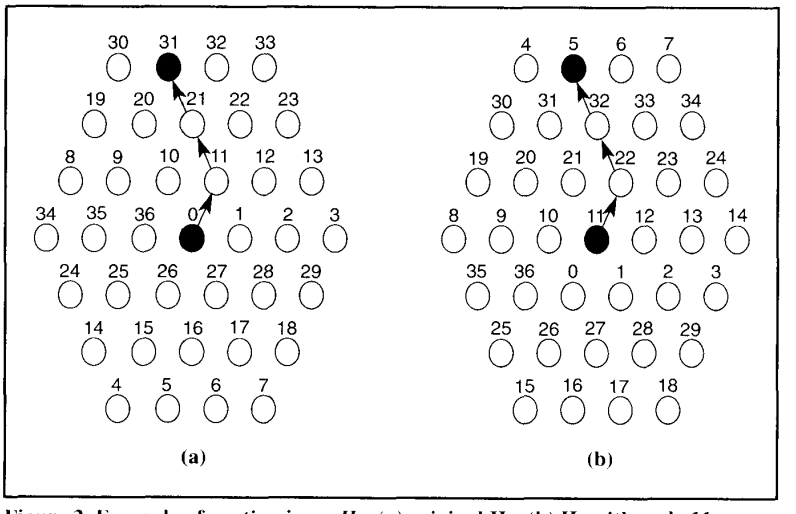

Figure 2. Example of routing in an H_4: (a) original H_4; (b) H_4 with node 11 placed at the center, $H_4(11)$.

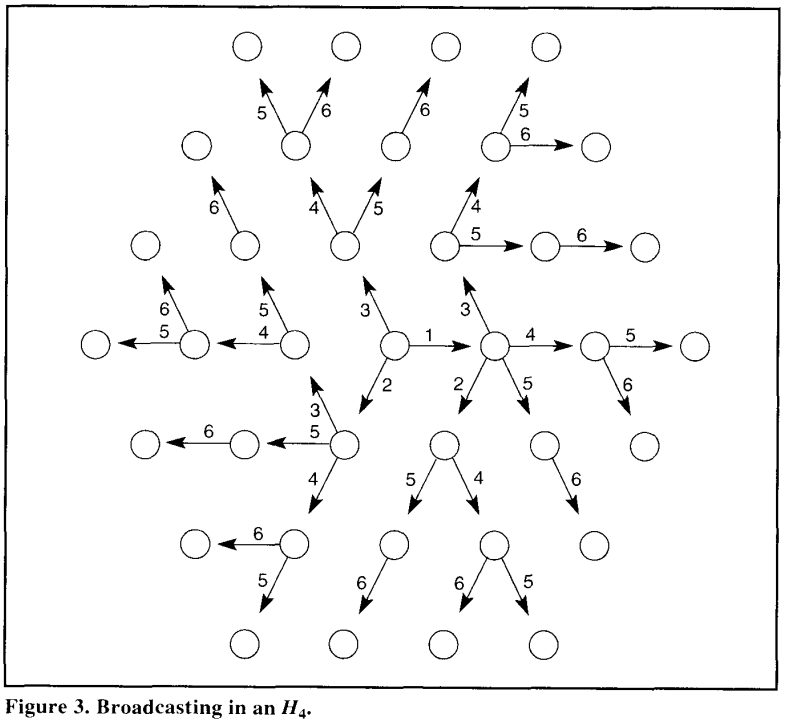

Figure 3. Broadcasting in an H_4.

Low-level architecture

We have developed special hardware support for time-constrained, fault-tolerant communications in HARTS, based on the addressing, routing, and broadcasting methods just described. Below, we discuss the need for extra hardware for communication processing, the main functional requirements of the NP, and a system-level architecture that realizes these functions.

Why communication hardware? Each node in a distributed system must be responsible for packet processing, routing, and error and flow control. Real-time applications impose additional functions related to meeting deadlines, time management, and housekeeping.

Packet processing can consume a substantial number of processor cycles and, in the absence of communication hardware, can deprive the host (node) of much needed computation power. In particular, the host is saddled with breaking a message into packets for transmission, constructing packet headers and trailers, framing packets, and calculating checksums. On reception of packets, the receiving host has to depacketize the message, strip headers and trailers, and compute the checksum for error checking. Each time a packet is transmitted or received, the host must be interrupted and context-switched to routines that perform these chores. This introduces substantial overhead because contemporary off-the-shelf processors are optimized to compute with register and cache data, which are lost in a context switch. For time-constrained, fault-tolerant communications, the host AP also has to handle several other functions that introduce significant computational overhead. These include message scheduling, route selection for reliable and timely delivery of messages, and clock synchronization.

All these functions divert significant computing power from time-critical applications. It is therefore necessary to offload such processing from the AP to special communication-processing hardware — that is, the NP.

Requirements of the network processor. Before designing and building the NP, we identified required functions, which must include efficient support for message processing, low-latency message transmission, and support for time-constrained, fault-tolerant communications. The operating system must establish deadline guarantees based on these functions.

Communication protocol processing. The NP's main function is to offload communication processing from the APs. When an AP needs to transmit a message, it provides the NP with information about the intended message recipient and the location of the message data. The NP's function is then to execute the operations necessary to pass the message data through the various layers of protocol down to the physical layer where it can be transmitted. In terms of the OSI (Open Systems Interconnection) reference model, the NP is responsible for functions from the transport layer down to the physical layer.

At the transport level, the NP establishes connections dependent only on the source and destination nodes, without concern for the route to be used. It also handles end-to-end error detection and message retransmission.

At the network level, the NP selects primary and alternate routes for establishing virtual circuits, forms data blocks and segments, and reassembles packets at the destination node. There are various switching methods, such as virtual cut-through switching, wormhole routing, store-and-forward packet switching, and circuit switching. Depending on traffic conditions in the network and the message type, the NP chooses an appropriate switching method for the message. The NP also detects and corrects errors at this level.

At the data link level, the NP provides access to the network for the messages. It performs framing and synchronization and packet sequencing. In addition to error checking at the network level, the NP performs checksum error detection and correction at this level.

Low-latency message transmission. Low communication latency is a key goal for NP design, and it influences task migration, task distribution, and load sharing. Latency impacts the system from application tasks down to hardware components. Because a significant portion of latency occurs in communication processing, achieving low-latency communications is intimately related to the implementation of communication protocols.

Support for time-constrained communications. The timely delivery of messages requires a global time base across the different nodes in HARTS. The NP is equipped with special hardware for clock synchronization and message time-stamping, providing the basis for the implementation of various real-time communication algorithms.

The NP also must support multiple interrupt levels to manage messages with different priority levels. The hardware must provide sufficient interrupt levels to give urgent messages priority over less urgent ones. Urgent messages must also have priority in the use of scarce resources such as message buffers and bandwidths. The NP must implement buffer management policies that maximize buffer space utilization while guaranteeing buffer availability to the highest priority messages. Similarly, if noncritical messages hold other resources needed by more critical messages, the NP must provide for resource preemption by the critical messages.

Another important NP function is monitoring the network's state in terms of traffic load and link failures. The traffic load affects the NP's ability to send real-time messages to other processors, while link failures affect system reliability. It is also possible for the NP to track its host's (or hosts') processing load and use the information for load balancing, load sharing, and task migration.

NP architecture. The NP architecture must support the functions just discussed. Although the HARTS NP architecture is similar to other communication architectures,[6] it has new features to facilitate real-time fault-tolerant communication. At the same time, it attempts to cost-effectively minimize message latency by intelligent management of messages and buffer memory.

The NP has five major components: the interface manager unit (IMU), the packet controller (PC), the routing controller (RC), the buffer memory, and the application processor interface (API), interconnected as shown in Figure 4. (The bus management unit and page management unit are auxiliary components.)

The API moves data between the NP and the host-node APs, while the RC moves data between the NP and the network. Within the NP, the IMU is the main processor that controls the movement and processing of message data. The buffer memory acts as a staging area for data to be transmitted to, or received from, the network, and for message data that must be temporarily stored at the node due to unavailability of outgoing links to the next node on the route to its destination. The RC implements the physical layer protocols for accessing the network and routing data to the node's neigh-

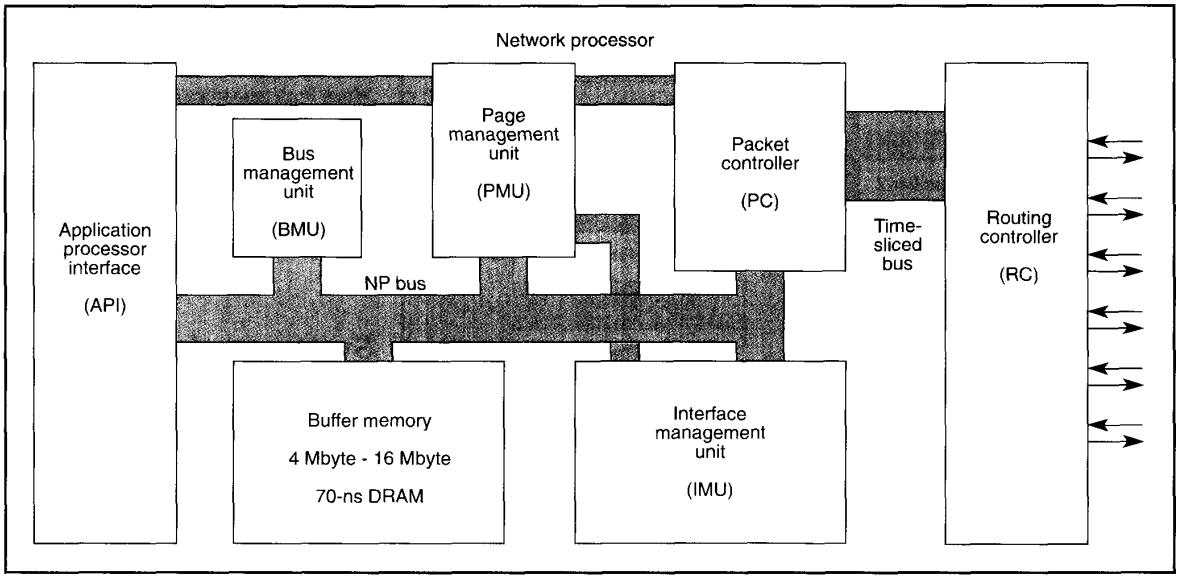

Figure 4. Block diagram of the network processor.

bors. It also supports virtual cut-through and wormhole routing by moving a message from an incoming to an outgoing link without buffering the message at the NP. Finally, the PC performs such functions as checksumming, packet framing, and deframing.

Interface manager unit. The IMU packetizes and depacketizes messages, schedules messages with different levels of priority, decides on switching methods based on message priority levels and network state, monitors the network state, performs error correction and message acknowledgment, and implements various real-time communication algorithms. Ease of software and hardware development and support, and availability, make a general-purpose RISC processor a reasonable choice for the IMU.

The IMU must provide multiple levels of interrupts and a short context switching time. To minimize message latency, the IMU must respond quickly to host requests for message transmission or reception services. The register window schemes in a typical RISC processor allow fast context switches, thus meeting this requirement.

The IMU has memory that can be used to store code and data. It also has access to the buffer memory, the staging area for messages being moved between the host and the network. To avoid excessive copying, the buffer memory usually serves as the IMU's data memory. Hence, the buffer memory is part of the IMU's address space.

Buffer memory. The buffer memory consists of RAM for the buffers and a buffer management unit. It stores messages waiting to be transmitted to or from the current node, and it acts as a temporary storage area for messages being routed through the current node. The amount of memory needed, usually only a few megabytes, is determined by the usage patterns of the application tasks.

The word size is 32 bits. With current DRAM access speeds of 70 nanoseconds, this gives a memory bandwidth as high as 457 megabits per second. This bandwidth is sufficient for access by the RC, the API, and the IMU, and for refresh cycles. Therefore, expensive static RAM or multiport memories are unnecessary.

The buffer manager arbitrates between the IMU, the API, and the PC for access to the buffer. It also handles buffer memory refresh by periodically accessing rows in the DRAM. The access priorities given to these different sources can be static, dynamic, or random, depending on the buffer management policy adopted.

Another function of the buffer manager is to provide addresses of free buffers for storing incoming packet data and to determine the location of packets ready for forwarding to an outgoing link. In other words, the buffer manager keeps the list of free buffer pages and tracks the location of various messages stored in the buffer. In instances where a message or packet spans more than a single page, the buffer manager keeps track of linked pages. The buffer management policy for the free list and the buffer allocation policy can be implemented with a separate microcontroller or the IMU.

Packet controller. The PC functions as a DMA (direct memory access) interface between the RC and the buffer memory, providing the IMU with inbound and outbound channels on which to transmit messages from or receive messages into the buffer. It accesses the buffer memory through the arbitration block of the buffer manager and transmits and receives messages without the IMU's intervention.

In transmitting and receiving packets, the PC performs the function of transparently framing and deframing packets. It does this by adding start-of-packet and end-of-packet bytes to the data bytes and computing the checksum as a packet is being sent. On reception of packets at the destination NP, the PC removes the packet header and trailer and computes the checksum to detect transmission errors. The detection of errors is signaled to the IMU via an interrupt, to trigger an appropriate recovery procedure.

Another function of the PC is to time-stamp messages as they are received and

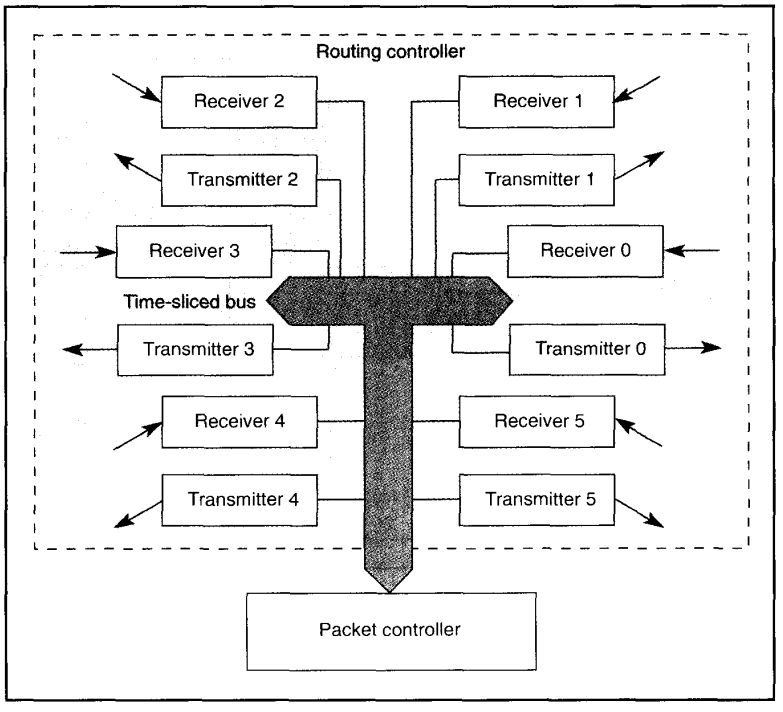

Figure 5. Block diagram of the routing controller.

transmitted. As will be discussed later, hardware time-stamping support is crucial to clock synchronization. The time stamp is appended to the message before the checksum bytes.

Routing controller. The RC is the interface between the NP and the network. It implements the physical layer and part of the data link layer. As Figure 5 shows, the RC consists of six receiver-transmitter pairs connected to the buffer manager and IMU through a time-sliced bus. The transmitters convert outgoing data into serial form for transmission on the outgoing serial line. Correspondingly, the receivers convert incoming serial data into parallel form and forward the data to a transmitter for onward transmission, in the case of virtual cut-through or wormhole routing, or to the buffer manager, if the data is to be stored in the current node. A single half-duplex serial line connects each receiver to a transmitter in a neighboring node.

A distinct feature of the RC is that the receivers can be microprogrammed to implement various routing algorithms used in HARTS. Various switching methods can also be programmed simultaneously into the RC and used selectively, on the basis of the type of messages being sent through the node and the network traffic at any particular time. This allows giving the highest switching and routing priority to critical messages, while optimizing the overall latency of other types of messages.

AP interface. The interface between the NP and the host APs is a VME bus. Data copying between a host AP and the NP is done by the API, which is a DMA interface to the VME bus. There are two ways of designing this interface for data transfer: mapping the NP's data memory into the host address space or copying data from the host's data memory to the NP's data memory. Mapping the NP into the address space of the host is may appear efficient, since it avoids the overhead of a system call. However, this mapping requires dedicated memory management hardware and kernel support for mapped address spaces, and it also incurs the overhead of data access over the VME bus. Depending on the typical size of the messages, burst-mode DMA transfer from the host memory to the NP memory may be more efficient.

In the burst-mode DMA transfer, the host initiates data transfer to the NP by writing to an API control register a pointer to the data in the host, as in a typical DMA sequence. The API then contends for the host VME bus and the NP buffer memory. When both resources are acquired, it copies the message data in burst mode directly from the host to the NP buffers. Upon completion of the transfer, the IMU is notified, and communication processing can begin. A similar sequence of operations is performed in reverse order for message receipt.

System evaluation

We have evaluated HARTS, using modeling and simulation with actual parameters derived from our implementation. Specifically, we examined how different switching methods can be combined to yield low latency. First, we evaluated the performance of virtual cut-through switching by developing analytic models and a low-level, event-driven simulator. Then, we compared virtual cut-through switching and wormhole routing.

Modeling and simulation of virtual cut-through. Since real-time applications normally require short response times, simple store-and-forward switching schemes are not suitable for HARTS. Hence, it supports fast switching methods such as virtual cut-through[7] and wormhole routing.[8] In virtual cut-through, packets arriving at an intermediate node are forwarded to the next node in the route without buffering if a circuit can be established to the next node.

Kermani and Kleinrock did a mean-value analysis of virtual cut-through performance for a general interconnection network.[7] However, a mean-value analysis is inadequate for real-time applications because worst-case communication delays often play an important role in real-time system design. A mean-value analysis cannot, for example, answer these questions: What is the probability of a successful delivery given a delay? What is the delay bound such that the probability of a successful delivery is greater than a specified threshold?

Kermani and Kleinrock wanted to avoid any dependence on the interconnection topology in their analysis. As a result, they assumed that the probability of packet buffering at an intermediate node is a given parameter. Since a reasonable estimate of virtual cut-through performance cannot be obtained without an accurate estimate of buffering probability, their approach becomes useful only if one can accurately

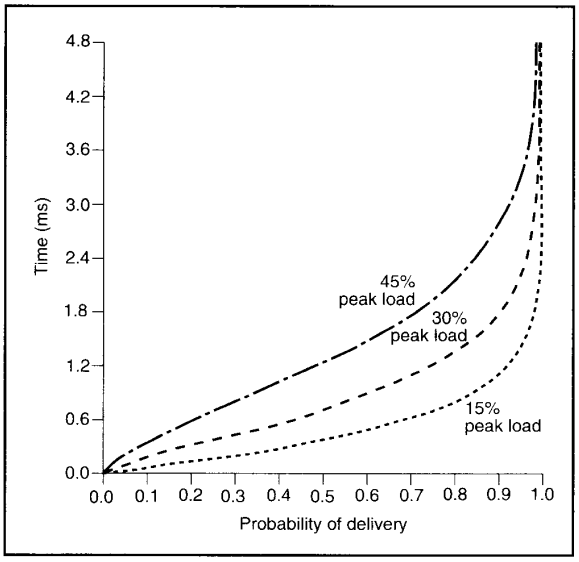

Figure 6. Delivery time versus probability of successful delivery.

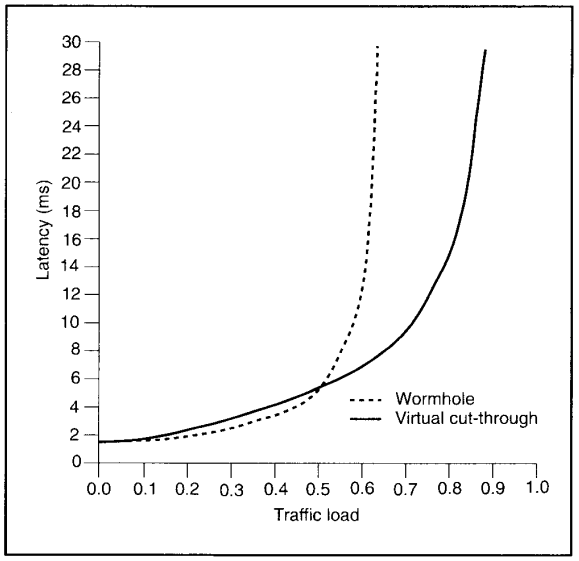

Figure 7. Latencies of wormhole routing and virtual cut-through switching for an H_5.

determine buffering probability for a given interconnection topology. This determination is not a simple matter, because each node in a distributed system handles not only all packets generated at the node, but also all packets passing through the node (transit packets). Consequently, to evaluate the probability of buffering, we have to account for the fraction of packets generated at other nodes that pass through each given node.

In contrast to Kermani and Kleinrock, we first derive the probability that a packet is destined for a particular node by characterizing the H-mesh topology. We use this probability of branching as a parameter in a queueing network to determine the throughput rate at each node in the mesh. Once the throughput rates are found, we can derive the probability that a packet can establish a cut-through at an intermediate node. From these parameters, we finally derive the probability distribution function of delivery times for a packet traversing a specified number of hops.

Figure 6 plots the inverse of the probability distribution function for a message traveling five hops. The three curves show the variation in the inverse of the probability distribution function for different message-generation rates or network traffic. These curves are useful in determining design parameters such as delay bounds. For example, one can select a delay bound such that the probability of message delivery within that bound is greater than a specified threshold. This would provide a probabilistic measure on the guarantees provided for real-time system operation.

In contrast to the analytic model, a simulator makes very few simplifying assumptions in modeling the behavior of virtual cut-through in HARTS. The simulator accurately models the delivery of each message by emulating the timing of the routing hardware along the packet route at the microcode level. It also captures the internal bus access overheads that the packets experience if they are unable to cut through an intermediate node. The simulator's detailed timing and tracking of messages allows investigation of various message scheduling strategies, access protocols, and memory management strategies. The simulator can also use any discrete distribution of packet lengths for which the user specifies the number, length, and probability of the different types of message. The simulator has been used to check the validity of analytic models by evaluating the HARTS communication subsystem under various realistic settings.

Evaluation of hybrid routing schemes. The basic idea of wormhole routing is that if a channel is not available, a message waits for it. Because the message is not removed from the network, it retains all resources from its source to the node at which it is waiting. Wormhole routing can be thought of as incrementally establishing a route because it does not surrender the resources it has acquired along the path from source to destination. One benefit is that the message need not reacquire resources once it has acquired them. Deadlock-free algorithms based on wormhole routing have been proposed by Dally and Seitz.[8]

Virtual cut-through differs from wormhole routing in that it stores the message at the node where it is blocked and releases the resources acquired on the path from the source to the blocking node once the message has been stored.

The advantage of both wormhole routing and circuit switching is that they guarantee delivery once a source-to-destination connection has been established. Virtual cut-through, however, can lower latency when the hogging of links due to wormhole routing and circuit switching worsens the congestion in the network.

To show the difference in performance of wormhole routing and virtual cut-through switching, we plotted their message latencies in Figure 7. For low traffic loads, wormhole routing takes less time on average to deliver messages; the opposite is true for high loads. The traffic load break-even point decreases as mesh size increases, because the average message distance increases with mesh size. Which routing method is more advantageous depends on the traffic load and average message distance. The routing controller described

earlier has the flexibility to dynamically select the better of the two methods.

Fault-tolerant routing

One attractive feature of point-to-point networks is their ability to withstand link and node failures. Exploiting this feature requires developing algorithms and providing mechanisms that preserve network communication in the presence of component failures. In this context, one must address correct routing of messages when one or more mesh components fail. This is of particular importance when the mesh is large and component failures thus are more likely, and when the system is expected to operate for long periods without maintenance. The ideal fault-tolerant routing algorithm would route messages by the shortest fault-free path, would require no extra hardware, would not cause unnecessary delays at intermediate nodes, and would quickly determine whether a destination was unreachable. The algorithm presented by Olson and Shin[4] comes close to meeting these criteria and requires each node to know only the condition (faulty or nonfaulty) of its own links.

Each node of an H-mesh can be seen as the convergence point of three axes, and the shortest path between two nodes can be expressed as offsets along no more than two of the three axes. Since each of the six links represents movement along one of these three axes, either in the positive or negative direction, fault-free routing can be accomplished by forwarding messages along links that will bring them toward zero. Our idea is to not interfere with this process until the message finds its path blocked. A message is routed by the fault-free algorithm until it reaches a node where all the links through which the message would ordinarily be forwarded (called the optimal links) are faulty. At that point the fault-tolerant algorithm intervenes.

At the point of message detouring, routing control is split between the fault-free algorithm and the fault-tolerant algorithm. A single bit in the message header determines which algorithm is currently making routing decisions. If this bit is clear, the message is in free mode, and the fault-free algorithm does the routing. Otherwise, the message is in detour mode, and the fault-tolerant algorithm does the routing. The fault-tolerant algorithm remains in control until it believes it has bypassed the faults that blocked the message path.

The fault-tolerant algorithm can be viewed as a simple wall-following algorithm. The message travels around the edge of a cluster of faults until it reaches the other side. Implementation is simple. When the optimal links are found to be faulty, the message is placed in detour mode, and the NP looks for nonfaulty links, starting with the link immediately counterclockwise of the optimal links and proceeding counterclockwise. The message is sent out on the first nonfaulty link found. If a message arrives at a node already in detour mode, it is sent out on the first nonfaulty link counterclockwise of the link by which it arrived. While in detour mode, the offsets to the destination are continually recalculated, and the message leaves detour mode when the distance to the destination is less than it was when the message entered detour mode.

As an example, consider the situation in Figure 8. A message has arrived at node 18, with node 1 as its eventual destination. At 18 the only optimal link is the one to node 0, which has failed; the message is placed in detour mode and sent to node 7. At 7 the fault-tolerant algorithm first tries to send the message to node 0, then to node 8, but finally must send it to node 15. At 15 the message is immediately forwarded to node 8. At 8 the message returns to free mode as node 8 is closer to node 1 than node 18 is. The message then completes routing normally.

An unreachable destination is revealed by the presence of a cycle. If the fault-tolerant algorithm cannot get the message to the destination, the message will cycle. Unfortunately, for certain classes of fault configurations, called incisions, the message will cycle even though the destination is reachable. Simulation results show that this type of fault is rare, occurring only with large numbers of faults. It can be dealt with at a cost of increased complexity in the routing algorithm. Strategies for detecting and routing in the presence of incisions are outlined by Olson and Shin.[4] They show that the H-mesh is extremely robust: If 50 percent of the links in an H_3 are faulty, a randomly chosen destination is reachable with probability greater than 0.95.

Clock synchronization

Widely recognized as one of the important requirements of a distributed real-time system, a global time base simplifies the solutions to design problems such as checkpointing, interprocess communication, and resource allocation.[9]

Central to the establishment of a global time base is the synchronization of the local clocks on different nodes in the system. Both hardware and software solutions to this problem have been proposed. The software solutions are flexible and economical but require the exchange of additional messages solely for synchronization.[10] The overhead imposed by these additional messages could be substantial, especially if a tight synchronization between processes is desired. Hardware solutions, on the other hand, require additional hardware at each node of the distributed system. They can achieve very tight synchronization between processes, with very little time overhead, but they require a separate network of clocks that is usually different from the network between the nodes.

For HARTS, we use a software solution that requires minimal hardware support at each node.[11] It is based on the interactive convergence algorithm given by Lamport and Melliar-Smith.[10] (Note, however, that any other software clock synchronization algorithm can be used for our scheme.) The algorithm assumes that the clocks drift apart only by a bounded amount during each resynchronization interval, R, during which each process reads the value of every process's clock. If the value of a clock read differs from its own clock by an amount greater than a threshold, the process replaces that value with its own clock value. The process then computes the average of all such values and sets its own clock to this average. Lamport and Melliar-Smith show that this algorithm can achieve synchronization and requires $3m+1$ processors to tolerate m faults.

Three major problems arise when this algorithm is used in a distributed system with a point-to-point interconnection network. First, it is difficult for a process to read the clock of a process to which it is not directly connected. Second, the message received by a process may be corrupted by a faulty intermediate process. Third, a queueing delay for the clock messages may cause a substantial difference between the real times at which a clock value is sent and received. Therefore, subtracting the clock value in the received message from the current clock value will not reflect the actual skew between the clocks of the sending and receiving processes. This problem is aggravated when the clock message must pass through multiple intermediate nodes.

Ramanathan, Kandlur, and Shin[11] solve the first problem by letting each process broadcast its clock to all processes at a specified time, with respect to its own local

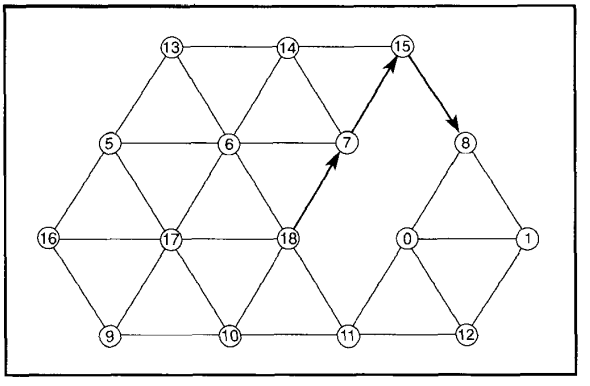

Figure 8. Example of fault-tolerant routing.

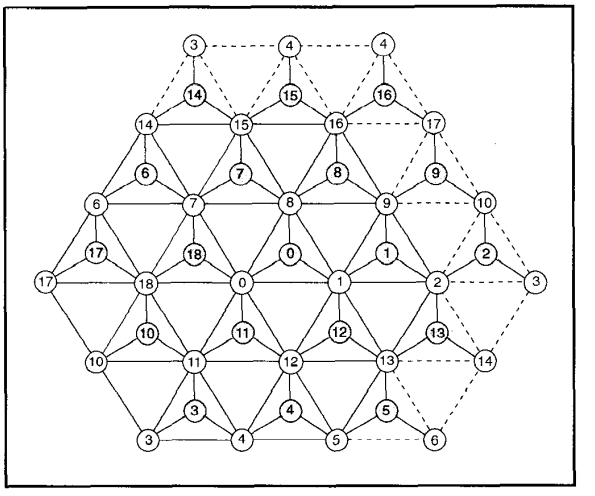

Figure 9. I/O controller placement.

clock, in the resynchronization interval. The second problem is eliminated by a broadcast algorithm that delivers multiple copies of the message to all processes through node-disjoint paths. For the third problem, it is not the size of the delay, but the fact that it is not known, that affects the clock skew. The message delivery time for clock messages is obtained by requiring each intermediate process to append to the message the delay incurred at that process.

The accurate computation of this delay needs some hardware support. There is some uncertainty in determining time of receipt because of a variable delay between the processor's receiving notification of arrival and actually "seeing" the message. Also, to compute the time delay within the node, the processor must have control on the exact time at which a message is transmitted on a link. These potential errors in estimating the time delay limit the accuracy with which we can compute the clock skew. This in turn affects the clock skew achievable with the synchronization.

To alleviate this problem, we use a hardware time-stamping mechanism at the link level for clock messages (see earlier section on the packet controller). When a link receiver detects a clock message, it appends a receive time stamp to the message. Similarly, when a clock message is transmitted, the link transmitter appends a transmit time stamp. At an intermediate node, the receive and transmit time stamps use the same local clock, so their difference gives a very accurate estimate of message time in that node. By computing the difference at intermediate nodes, we can keep the total number of time stamps down to five and prevent message length from growing as network size increases.

For any synchronization algorithm, R is a function of the maximum clock skew desired. R decreases with the desired maximum skew and becomes negative for small values. From a practical viewpoint, overheads for the synchronization algorithm increase as R decreases, so it is desirable to have R as large as possible. This function effectively determines the type of skew that can be achieved for the system with a particular synchronization algorithm. The derivation of this function for the synchronization algorithm described here is given by Ramanathan, Kandlur, and Shin.[11] This algorithm can achieve moderately tight synchronization. For example, in an H_3, a maximum clock skew of 100 microseconds can be achieved using R=6.23 seconds.

I/O architecture

Most work on distributed computing systems has centered on interconnection networks, programming and communications paradigms, and algorithms. However, little has been done specifically about the I/O subsystem in a real-time environment, despite its obvious importance. Clearly, a real-time computer can process data no faster than it can acquire the data from sensors and operators. Note that I/O devices in a real-time environment are sensors, actuators, and displays, whereas they are magnetic disks and tapes for general-purpose systems. Due to the distinct timing and reliability requirements of real-time applications, solutions suited to general-purpose systems are not usually applicable to the real-time environment.

To avoid the accessibility problems of nondistributed I/O, I/O devices need to be distributed and managed by relatively simple, and reliable, controllers. Morcover, to improve both accessibility (and thus reliability) and performance, there must be multiple access paths (called multiaccessibility or multiownership) to these I/O devices.

I/O interconnection architecture. I/O devices are clustered, and a controller manages access to the devices of each cluster. The I/O controller (IOC) can be simple since HARTS uses simple data links to the computation nodes. The IOC need only handle sending and receiving simple messages via a set of full-duplex links, not providing virtual cut-through capabilities and other features of a full-blown NP. To keep the IOCs and the I/O links down to a reasonable number, the number of IOCs is restricted to no greater than the total number (p) of computation nodes in the mesh. This has certain benefits for one of the management protocols explained later.

Having established the potential number of I/O nodes, we need to decide how many nodes each IOC will be connected to. If the maximum number (p) of IOCs are assumed to exist in an H_3, for example, then Figure 9 shows a logical connection scheme.[12] Each IOC can be thought of as being in the center of one of the upward-pointing triangles in this figure; the IOC is connected to each of the nodes that make up the triangle, called its left, right, and upper partners. This gives three possible avenues of access to each IOC. Note that if the maximum

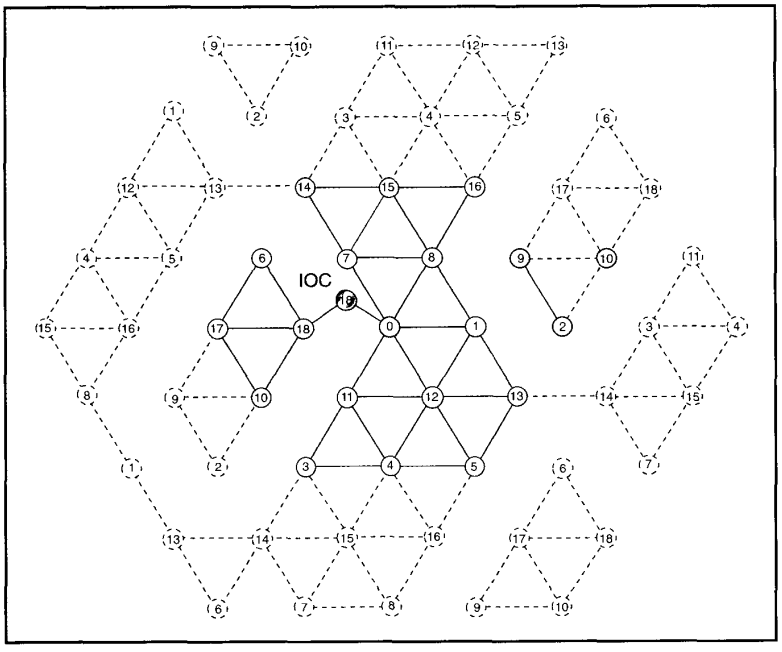

Figure 10. Unreachable static owner.

number of IOCs are used, the number of I/O links required is equal to the number of standard communication links, or $9e2–9e+3$ for an H_e. One could similarly place IOCs at the (logical) center of the downward-pointing triangles as well, allowing for up to $2p$ IOCs, but this would double the maximum possible number of I/O links required and disturb certain homogeneous effects of limiting the number of IOCs to the number of nodes.

Management protocols. The desire for simple I/O controllers presents a problem in HARTS, because the natural tendency would be to assign sensors and actuators, both relatively complex and expensive devices, to individual nodes or NPs and use the given interprocess communication (IPC) channels in HARTS to handle the I/O traffic. We can still use the given IPC channels, but instead of permanently tying down a given I/O device to one node, we allow several nodes to communicate with each I/O device. There are two fundamentally different protocols for managing this communication.

The first management protocol, the static protocol, assigns one node to each IOC as its owner, but with the important provision that the owner can be changed if the original owner becomes faulty. In this protocol, one of the IOC links is defined as the active link, and the rest remain inactive as spares. The second, dynamic protocol, allows the IOC owner to be defined dynamically, providing greater accessibility and requiring fewer hops on average to reach the IOC owner. In this protocol, the IOC decides which link will be active at any given time.

Figure 10 is an example in which a process in node 13 wants service from IOC 18, but since node 18 is the owner under the static protocol and is not reachable from 13, it cannot obtain service. If node 0 were the owner instead of 18 — which is possible under the dynamic protocol — it could be serviced.

In addition to making IOCs accessible where static ownership would make them inaccessible, the dynamic protocol takes into account the fact that one partner may be closer to a node requesting service than the other partner. Since this protocol chooses the closest of the partners that respond, the I/O traffic may have fewer hops to travel. However, its disadvantages are that it is more difficult to implement and involves arbitration overhead after servicing each I/O request. It may also be undesirable because there is no single node through which all I/O requests will travel and which could perform some I/O management tasks. Shin and Dykema give a comparative analysis of these two protocols.[12]

All the high-level architectural issues of HARTS have been resolved, and the lower-level components are being designed or implemented. The routing controller, a key component for fast switching, has been fabricated, and its testing is almost complete. The packet controller, the second generation of the routing controller, and other NP components are currently being designed and simulated. In parallel with the architectural work, we are also designing and implementing a software communication subsystem for HARTS. The primary objectives of this subsystem are to deliver messages within certain deadline constraints, support mechanisms for group communication and reliable broadcasting, offer services such as maintenance of a global time base, and monitor system behavior. ■

Acknowledgments

The work reported in this article was supported in part by the Office of Naval Research under contract N00014-85-K-00122. The author would like to thank all current and former members of the RTCL for their contributions to the HARTS project. Specifically, Ming-Syan Chen developed wrapping, labeling, and routing techniques, James Dolter and Parameswaran Ramanathan designed and implemented the routing controller chip, Stuart Daniel and Teng-Kean Siew are currently working on the design and implementation of the network processor, Dilip Kandlur and Daniel Kiskis developed HARTOS, Alan Olson developed a fault-tolerant routing algorithm, and Greg Dykema played a key role in developing the I/O architecture. The author is also indebted to Andre van Tilborg, Gary Koob, and James Smith at the Office of Naval Research for their encouragement.

References

1. 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, July 1989, pp. 72-89.

2. K.S. Stevens, "The Communication Framework for a Distributed Ensemble Architecture," AI Tech. Report 47, Schlumberger Research Laboratory, Palo Alto, Calif., Feb. 1986.

3. M.-S. Chen, K.G. Shin, and D.D. Kandlur, "Addressing, Routing and Broadcasting in Hexagonal Mesh Multiprocessors," *IEEE Trans. Computers*, Vol. C-39, No. 1, Jan. 1990, pp. 10-18.

4. A. Olson and K.G. Shin, "Message Routing in HARTS with Faulty Components," *FTCS-19, Digest of Papers*, Computer Society Press, Order No. 1959, June 1989, pp. 331-338.

5. J.W. Dolter, P. Ramanathan, and K.G. Shin, "A Microprogrammable VLSI Routing Controller for HARTS," *Proc. Int'l Conf. Computer Design: VLSI in Computers*, Computer Society Press, Order No. 1971, Oct. 1989, pp. 160-163.

6. E.A. Arnould et al., "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM, New York, 1989, pp. 205-216.

7. P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, Vol. 3, 1979, pp. 267-286.

8. W.J. Dally and C.L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Computers*, Vol. C-36, No. 5, May 1987, pp. 547-553.

9. L. Lamport, "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Trans. Programming Languages and Systems*, Vol. 6, No. 2, Apr. 1984, pp. 254-280.

10. L. Lamport and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *J. ACM*, Vol. 32, No. 1, Jan. 1985, pp. 52-78.

11. P. Ramanathan, D.D. Kandlur, and K.G. Shin, "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems," *IEEE Trans. Computers*, Vol. C-39, No. 4, Apr. 1990, pp. 514-524.

12. K.G. Shin and G.L. Dykema, "Distributed I/O Architecture for HARTS," *Proc. 17th Int'l Symp. Computer Architecture*, Computer Society Press, Order No. 2047, June 1990, pp. 332-342.

Kang G. Shin is a professor and associate chair of electrical engineering and computer science at the University of Michigan, Ann Arbor. From 1978 to 1982 he was on the faculty of Rensselaer Polytechnic Institute. He has authored or coauthored more than 180 technical papers on fault-tolerant computing, distributed real-time computing, computer architecture, and robotics and automation. In 1987 he received the Outstanding Paper Award from the *IEEE Transactions on Automatic Control* for a paper on robot trajectory planning. In 1989 he received the Research Excellence Award from the University of Michigan.

Shin received the BS degree in electronics engineering from Seoul National University, South Korea, in 1970, and the MS and PhD degrees in electrical engineering from Cornell University in 1976 and 1978, respectively. He is a distinguished visitor of the IEEE Computer Society.

Readers may write Shin at the University of Michigan, Real-Time Computing Laboratory, Dept. of Electrical Engineering and Computer Science, Ann Arbor, MI 48109-2122.