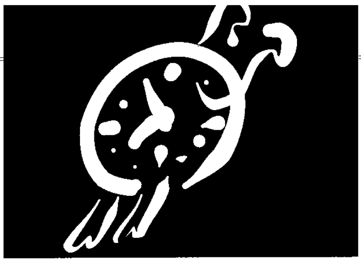


# A Distributed Real-Time Operating System

KANG G. SHIN, DILIP D. KANDLUR, DANIEL L. KISKIS, PAUL S. DODD, HAROLD A. ROSENBERG, and ATRI INDIRESAN, University of Michigan

◆ *An enhanced uniprocessor kernel combines with a set of evaluation tools that let you experiment with the performance and fault tolerance of real-time programs.*

Digital computers are now commonly used for such embedded real-time applications as computer-integrated manufacturing, industrial process control, defense systems, and electric power distribution and monitoring. These applications usually impose stringent timing and reliability requirements because a system failure could lead to catastrophe.

Distributed systems with point-to-point interconnection networks are natural candidate architectures for these applications primarily because they are most likely to meet stringent timeliness and reliability requirements. The key to success in using a distributed system for real-time applications is the timely execution of computational tasks, which usually reside on different nodes and intercommunicate to accomplish a common goal.

Providing deadline guarantees for real-time tasks is very difficult because it involves interaction among architectures, operating systems, and tools to evaluate both performance and dependability. To study these interactions, we have been designing, implementing, and evaluating a 19-node hexagonal mesh, called Hexagonal Architecture for Real-Time Systems. The HARTS architecture is briefly described in the box on pp. 60-61.

In this article, we describe two versions of the HARTS operating system, which is based on Software Components Group's pSOS uniprocessor kernel. In one version, we have enhanced pSOS services to provide interprocessor communication and a distributed naming service. In the second version, we add real-time fault-tolerant communication, including reliable broadcasting, clock synchronization, and group communication.

We are also developing a suite of tools to evaluate the performance and dependability of HARTS, the HARTS operating system, and other real-time systems. The goal of these tools is to let HARTS users perform a broad range of experiments to measure the performance and fault-tolerance of various programs.

## HARTS OPERATING SYSTEM

HARTS operating system, HARTOS, is designed to exploit the network processor on each HARTS node. It consists of pSOS executing on the application processors, the HARTOS protocol code executing on the Ethernet processor, and an interface between them. Work on a customized network processor is underway to replace the Ethernet processor. Because the design will be the same when the custom network processor is complete, the "network processor" we refer to in subsequent description is the current Ethernet processor.

**pSOS node kernel.** The pSOS uniprocessor kernel serves as the executive on each application processor and provides facilities for process and memory management, event handling, and interprocess communication. A process is the unit for sequential execution, resource ownership, and scheduling. Processes are named and use a name to locate a process they wish to communicate with. The kernel uses a preemptive priority-based scheduler that lets processes of equal priority cycle in a round-robin fashion. Processes can dynamically change their priority and protect themselves from interruption while in critical sections by setting their mode to nonpreemptive. Processes communicate with each other primarily through message exchange, an object that holds a queue of messages or processes, to allow many-to-many process communication.

**Version 1.** In the first version of HARTOS, we extended the pSOS kernel to work in a multiprocessor and distributed environment. New services include interprocessor communication (both datagram and remote procedure calls) and a distributed naming service. We built this version atop Communication Machinery's K1 kernel, which provides protocol-support facilities such as the Ethernet interface and mechanisms for setting time-outs and handling interrupts.

HARTOS extends the pSOS process control, interprocess communication, time queries, and name-lookup functions to work between processors on the same node and across the network. Calls may be blocking or nonblocking, and the user can specify a time-out and maximum number of retries to be used for each remote call. Nonblocking calls are not queued. This policy is consistent with the requirements of many real-time applications, such as control applications in which sensor data is gathered periodically and only current data is of value. A function is available to let the sending process block until an outstanding operation completes.

**Processes communicate primarily through message exchange, an object that holds a queue of messages or processes.**

pSOS provides exchanges for passing short messages between processes on the same processor. Shared memory may be used to pass larger data blocks. An extension of pSOS message-passing primitives would still handle only small messages, so we needed a mechanism for large data transfers across the network. Version 1 includes operations that transfer up to 16 Kbytes of data between processes and a mechanism for prioritizing data transfers to order operations from different processes. Providing a mechanism to assign priorities for other operations was not a concern, since these operations have few resource and timing requirements, and version 1 does not support timeliness guarantees.

We specify destinations for system calls using an internal address that consists of an application processor ID and an exchange or process ID. The HARTOS naming service provides the addresses. The network processors maintain a distributed name table, which is used to map logical names to internal addresses; the application processor enters names explicitly into the table. Each network processor maintains a table of map entries for entities declared on that node by the associated application processors.

A process may locate a named server by submitting a find request to the network processor. During a find operation, the network processor searches the local name table for a match. If there is no local match, it broadcasts a request for a name mapping to other network processors on behalf of the process that made the request. Only network processors that detect a match reply to the request, and the first reply received is taken. The request is handled entirely by the network processors and is not visible to the application processors.

**Version 2.** As the communication services became more complex, we found we needed a more sophisticated environment for implementing and experimenting with different communication protocols. In the second version of HARTOS, we are adding the communications subsystem shown in Figure 1. The additional real-time communication services — which contain some components that are different implementations of version 1 components — are a global time base and a real-time channel service to provide communication with guaranteed delays. The global time base, which we plan to maintain using the clock-synchronization algorithm described by Parmesh Ramanathan and colleagues,[2] enables the real-time channel service. Version 2 will also include fault-tolerant services such as reliable broadcasting and group communication.

Another difference between the two versions is that we are using a derivative of the *x*-kernel as the executive for the network processor in version 2. The K1 kernel provides only primitive support, and the *x*-kernel has proved effective in supporting communication protocols.[1] Applications can be implemented as protocol modules that can be integrated in the protocol stack. The $x$-kernel provides several facilities for implementing protocols, such as a uniform protocol interface and libraries, to efficiently manipulate messages and maintain mappings. It also comes with utilities to configure and test different protocol stacks.

**Communication subsystem.** Communication services are provided in the form of protocols running on the $x$-kernel. Figure 1 shows the protocols' interdependencies. At the lowest level is the link-level protocol, which uses a multiclass earliest-due-date scheduling algorithm[3] to support a mix of normal (best-effort) and real-time (guaranteed maximum latency) traffic. The normal link-level protocol also supports a reliable broadcast mechanism. The figure shows the real-time-channel link level as logically distinct from the normal link level mainly because the two differ in how they manage the buffer and in-transit messages.

At the next level up are the clock-synchronization protocol — which maintains a system-wide synchronized time base, accessible by application tasks and other protocol modules — and the Frag protocol — which fragments large messages into link-level-size packets and transports them. When Frag receives the message fragments, it collects them and coalesces them into a single message.

The module for the clock-synchronization protocol interacts with its other nodes' clock-synchronization protocols to provide the global time base. The protocol relies on a hardware time-stamping mechanism to disseminate the clock value of a local node to other nodes[2] and uses the interactive convergence algorithm on the clock values it receives from other nodes.

The programmable routing controller, which is under development, affixes a transmit-time stamp to a clock packet just before its transmission and appends a receive-time stamp to any clock packet it receives. This hardware time-stamping ensures that the protocol can factor out delays in the processing and propagation of clock messages, so that they do not affect the tightness of the synchronization.

The protocol also uses a hardware-maintained local clock with a resolution of 1 ms. The clock, which can be read directly by processes running on the network processor, is used to set deadlines for messages and processes and to determine

**HARTS ENVIRONMENT**

Hexagonal Architecture for Real-Time Systems is an experimental distributed real-time system, consisting of multiprocessor nodes connected by a point-to-point interconnection network. Each HARTS node has several application processors, which run application tasks, and a network processor, which contains the interface to the network, buffer memory, and a RISC processor. The RISC processor handles most of the processing related to communication.

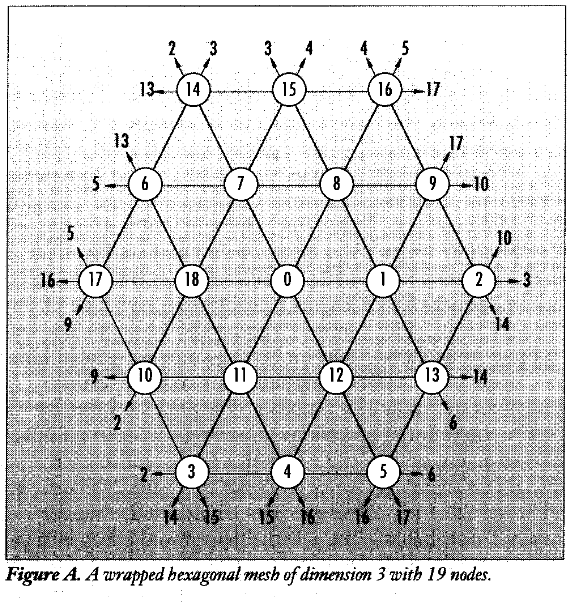


*Figure A. A wrapped hexagonal mesh of dimension 3 with 19 nodes.*

The HARTS interconnection has a continuously wrapped hexagonal mesh topology, which is a regular homogeneous graph in which each node has six neighbors. You can visualize the graph as a simple hexagonal mesh with wrap links added to the nodes on the periphery. Figure A illustrates the wrapping scheme for the peripheral nodes, in which the dimension of a hexagonal mesh is the number of nodes on a peripheral edge. The version of HARTS under construction is a hexagonal mesh of dimension 3 with 19 nodes.

We chose this wrapped hexagonal topology because of its potential to meet the timeliness requirement and high reliability — both of which are crucial for any real-time system design. It also offers scalability and is easy to construct.

The interconnection network of a distributed system must often connect thousands of homogeneously replicated processor-memory pairs, each of which is called a processing node. Processing-node synchronization and communication is done through message passing. The homogeneity of processor nodes and the interconnection network is important because you can replicate both hardware and software components inexpensively. Each processing node in the multiprocessor should have fixed connectivity so that the designer can use standard VLSI chips and communication software.

The interconnection network should have reasonably high connectivity to provide the alternative routes necessary to detour faulty nodes and links. Routing and broadcasting must be efficient to ensure high-performance task execution.

For structural flexibility, a

the order of services. The protocol provides control operations that let application processors' processes access the time.

The user datagram protocol at the next level supports the unreliable datagram service. The user-datagram protocol and all higher level protocol modules provide services that the user can access directly. These are reliable broadcasting, remote procedure calls, and a real-time- channel service.

***Reliable broadcasting.*** Although broadcasting is very simple for networks like Ethernet and token ring, in which every node sees a transmitted message, it is more involved for point-to-point interconnection networks. For this type of network, a simple nonredundant broadcast algorithm, which delivers a single copy of a message to every node, essentially constructs a spanning tree for the network graph rooted at the source node. The goal is to keep the spanning tree as short as possible by minimizing the number of store-and-forward communication steps. You can further reduce the required store-and-forward communication steps using the virtual cut-through switching scheme described by Parviz Kermani and Leonard Kleinrock.[4] We will provide such broadcasting through the HARTS programmable routing controller.

Reliable broadcasting is essential for implementing algorithms for clock synchronization.

Reliable broadcasting is essential for implementing algorithms for clock synchronization, distributed fault diagnosis, and distributed agreement when there are faults. A nonfaulty node must correctly deliver its private value to all other nonfaulty nodes, yet faulty nodes can discard, corrupt, and possibly alter the information passing through them. On-line distributed diagnosis schemes can help identify faults, but they do not always give total fault coverage. Therefore, broadcast algorithms must work even when you cannot identify all the faulty processors. In our algorithm, multiple copies of the message are delivered through disjoint paths to every sys-

system must also possess fine scalability — the processor nodes needed to increase the network's dimension by one.

To meet all these requirements, we considered several topologies, including hypercubes, square meshes, three-dimensional torus, hexagonal meshes, and octal meshes. The need for fixed connectivity and planar architecture for easy VLSI and communication implementation, finer scalability, reasonably high fault tolerance, and ease of construction made the hexagonal mesh the best choice. Ming-Syan Chen and colleagues give a detailed comparison of these topologies.[1] James Dolter and colleagues describe HARTS message-delivery performance.[2]

Figure B shows the current configuration of HARTS, with VMEbus-based nodes, each of which has one to three application processors, a system controller card, a network processor card, and an Ethernet processor card. The processor cards have a Motorola 68020 32-bit processor and 4 Mbytes of dual-ported RAM accessible from the VMEbus. The Ethernet processor card uses a 10-MHz 68000 processor and an AMD Ethernet controller device. The Ethernet links the development workstations. We are developing a custom network processor board that will replace the Ethernet board. The custom board will have a programmable routing controller chip to support high-speed switching mechanisms such as virtual cut-through. The chip will be used in the network processor card as the front-end interface to the hexagonal mesh.

**REFERENCES**

1. M. Chen, K. Shin, and D. Kandlur, "Addressing, Routing and Broadcasting in Hexagonal Mesh Multiprocessors," *IEEE Trans. Computers*, Jan. 1990, pp. 10-18.
2. J. Dolter, P. Ramanathan, and K. Shin, "Performance Analysis of Message Passing in HARTS: A Hexagonal Mesh Multicomputer," *IEEE Trans. Computers*, June 1991, pp. 669-680.

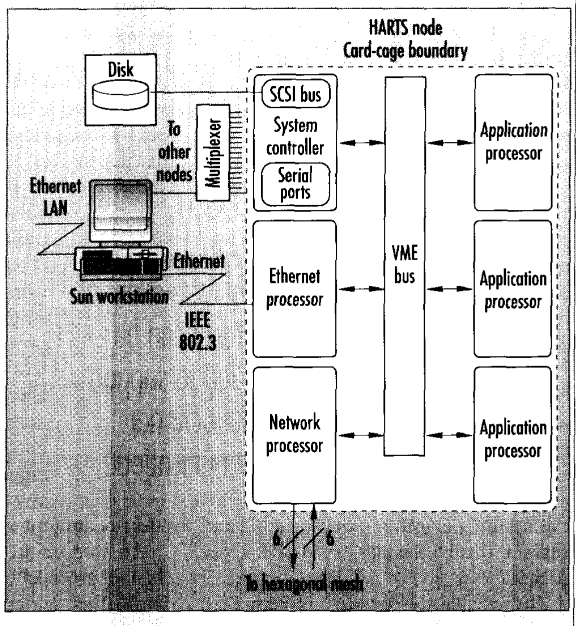


***Figure B.*** *The HARTS software-development environment and node architecture.*

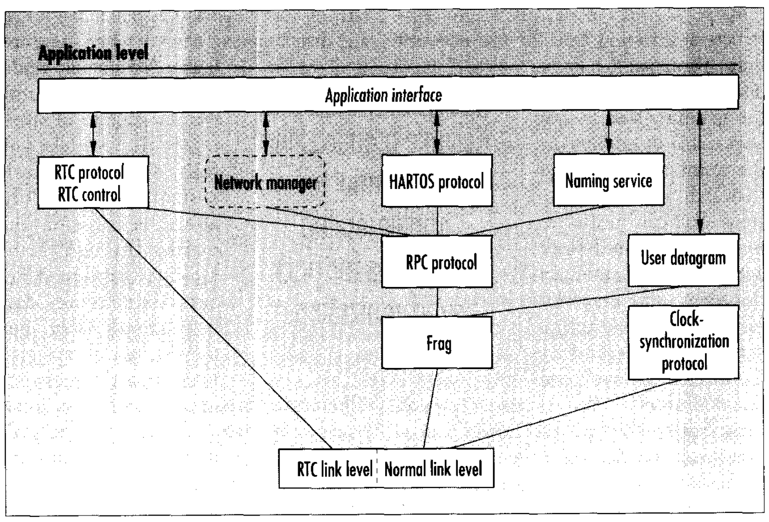


***Figure 1.*** *Simplified view of the communication subsystem in version 2 of the HARTS operating system.*

tem node. The receiving nodes identify the original message from the multiple copies using a scheme appropriate for the fault model, such as majority voting. The algorithm, called the *k*-reliable broadcast algorithm, delivers *k* copies of the message to each node through disjoint paths.[5] If encryption is used to guard against message corruption, a *k*-reliable broadcast will be resilient to the loss of *k*–1 message copies.

***Remote procedure calls.*** The protocol for remote procedure calls implements a request-reply operation with at-most-once semantics, using the technique described by Andrew Birrell and Bruce Nelson.[6] The protocol handles only the transport mechanism for remote procedure calls. Clients of this protocol are expected to marshal the call arguments into a request packet and extract return results from the reply packet.

To provide the services implemented in HARTOS version 1, we place a client protocol module atop the protocol module for remote procedure calls. Some of these calls are now redundant because their functions have been subsumed in other protocols. For example, the naming-service calls are handled by the naming-service protocol. Data-transfer operations can be similarly implemented using the user-datagram and frag protocols.

***Real-time-channel service.*** Unpredictable delays in message delivery can adversely affect the execution of the real-time tasks that depend on those messages. Making communication predictable is difficult, however, because network delays can be nondeterministic for many networks.

The real-time-channel protocol can be extended to support bidirectional communication.

Our real-time-channel protocol provides guarantees of timely message delivery. It handles the transmission and reception of both large and small messages but does not expect any acknowledgment and does not attempt to retransmit packets.

An important task of the real-time-channel protocol is to establish a channel. To do this, it must know the channel's source and destination, worst-case traffic patterns, and the maximum allowable delay in end-to-end message transit. Channel establishment is complex because the system must reserve resources at multiple nodes in the network, which is why making channel establishment a separate service is often the preferred approach. Our procedure establishes a route through the network from source to destination, ensuring that adequate communication bandwidth, buffer space, and processing bandwidth are available at all intermediate nodes in the path.[3] By centralizing this function, we can better use network resources because we can select routes appropriately to balance network load. This approach also makes it easier to handle network reconfiguration if the network fails.

In HARTOS, Network Manager establishes channels. Network Manager is present only on a special node or a set of nodes, which the dashed lines in Figure 1 indicate. We made Network Manager fault tolerant by assigning redundant copies of it on multiple nodes and maintaining consistency among these copies with atomic broadcasts.[3]

The protocol's control functions are handled by the real-time-channel control module, which uses a mechanism for remote procedure calls to contact Network Manager. When a channel is successfully established, Network Manager returns the selected route for the channel along with the worst-case delay for each link on the route. This information is used to set deadlines for messages belonging to real-time channels.

The real-time-channel protocol usually treats each send request as a separate message. It enforces a rate-based flow control mechanism and adjusts the time constraints of the message accordingly. It splits long messages into fragments and assigns them a common deadline. When receiving message fragments, the protocol collects them and delivers the total message to the client. If some fragments of a message do not arrive within the deadline, the partial message is delivered with an appropriate warning.

The real-time-channel protocol can be extended to support bidirectional communication (a real-time channel is a *uni*directional connection between two endpoints). Although we have not done this extension, it is a trivial task. The extended protocol creates a pair of real-time channels simultaneously in two directions. The parameters for these two channels are specified separately.

A drawback of the current real-time-channel protocol is that the real-time-channel service deals with one-to-one communication. While it provides guarantees of real-time performance, it does so only if there are no faults. We are considering alternatives to relax this restriction and provide a fault-tolerant real-time service. One option is to reroute a real-time channel if a component fails, but this could take more than the guaranteed delay. In addition, the failure of an application node would cause a total loss of service.

A more promising alternative is to extend the real-time channel concept to provide one-to-many communication. This extension could provide a group channel for replicated applications to communicate in real-time. The issues we must address in such an extension are multicast route selection (similar to broadcasting but with a subset of nodes), message consistency, and message ordering.

## EVALUATION TOOLS

We are developing three tools to evaluate the performance and dependability (fault tolerance) of HARTS hardware and software: a synthetic-workload generator, a monitor, and a fault injector. The generator produces a synthetic workload, the monitor collects the performance data, and the fault injector simulates faulty behavior for further study. Together these tools create a facility that lets the user perform a wide range of experiments. The tools are independent, so they are equally effective separately or together, depending on the requirements.

**Synthetic-workload generator.** Figure 2 shows the design of the synthetic-workload generator. The generator compiles a high-level description of a workload to produce a synthetic workload — programs that model the application programs' structure and behavior.[7] The synthetic workload is specified using the Synthetic Workload Specification Language.

As part of this compilation, the generator compiles a graph of the task and its parameters, performs correctness checking on the task graph, determines if the workload components are connected legally, and reports any errors.

The generator has two outputs: One is a set of data structures that describe the task graph and workload and experiment parameters. The other is the C code for the application tasks. These files are compiled and linked with the object code of the synthetic-workload driver and the library of synthetic operations. The result is an executable synthetic workload, ready to be downloaded to HARTS node processors. No intermediate user intervention is required.

The synthetic workload exercises the system's resources and thus accurately models how the application program would perform.

***Synthetic workload.*** A synthetic workload is a set of programs that execute on the target computer while the user measures performance. The structure and behavior of the programs that make up the synthetic workload model those of the application programs. Synthetic workloads are akin to rapid prototypes. The difference is that rapid prototypes demonstrate the functionality of the software from the user's viewpoint, while synthetic workloads demonstrate how the system sees the software's resource use. The synthetic workload exercises the system's resources and thus accurately models how the application program would perform.

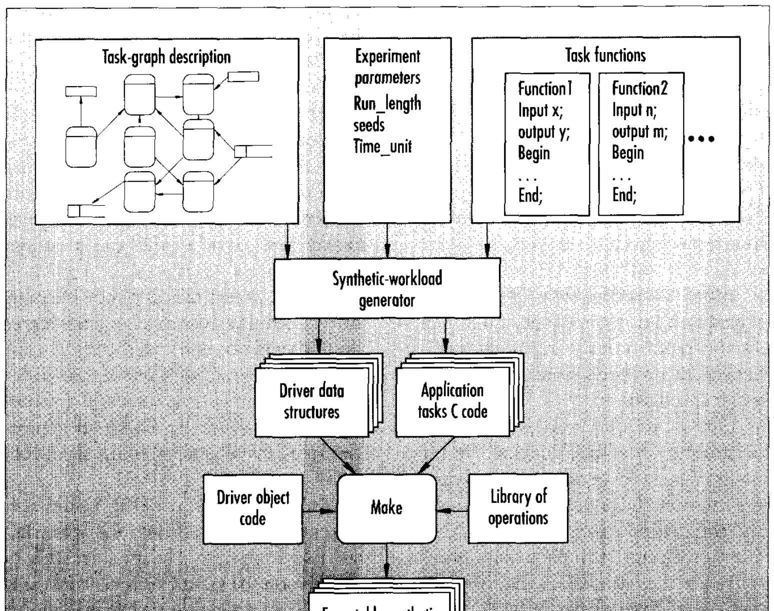
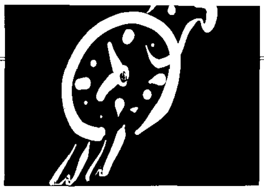


*Figure 2. Synthetic-workload generation.*



The user controls the workload's behavior and structure, and so is free to evaluate the system under many workload conditions. The tasks of the synthetic workload are parameterized, so that the user can easily change workload characteristics as needed.

Since HARTS is an experimental testbed, we expect it to be used to implement and evaluate a wide range of system features. Designers will want to evaluate the system under a range of workload conditions. A synthetic workload is ideal for this experimentation because it can represent either real workloads or specific, possibly anomalous, workload conditions.

The synthetic workload is composed of a driver and synthetic application tasks. The driver performs the control functions related to the experiment. Using the data structures produced by the generator, the driver creates, initializes, and activates all the objects in the task graph. It also communicates with the drivers on other processors so that control can be synchronous. The synthetic application tasks are responsible for creating the resource demands on the system. They execute during the experiment with little interference from the driver.

***Synthetic Workload Specification Language.*** SWSL is based on a number of structured-analysis and rapid-prototyping notations, enhanced to provide greater flexibility and easier specification.

One of its major roles is to specify parameters for the workload components. The tasks in our workload model, for example, have parameters that specify their execution and resource-use characteristics. Of particular importance are the parameters that specify the tasks' real-time characteristics, such as invocation periods, deadlines, and other real-time scheduling requirements.

SWSL also specifies the functions the tasks will execute. A function is a sequence of operations that exercise different system resources. In practice, these operations are taken from a library of synthetic operations. The library contains parameterized operations, each of which uses a specific resource. Synthetic operations may be interspersed with user-supplied C code to produce customized functions. SWSL also lets you specify control constructs within the function. Using these constructs, the synthetic tasks can simulate the data-dependent branching and looping behavior of actual application tasks. Through the combination of synthetic operations, control constructs, and user-supplied C code, SWSL lets users specify a wide range of task behaviors.

The language to specify synthetic workloads is based on structured-analysis and rapid-prototyping design notations.

We use this parameterized approach for two reasons. First, a high-level specification of the application software, such as the structured-analysis model, will generally be a good approximation of the workload's structure. Thus, by using a similar model, we can produce a synthetic workload that closely approximates the structure and behavior of the workload being modeled. Experimental evaluations performed using this synthetic workload should then provide useful and meaningful results.

Second, as real-time software becomes more complex, the use of structured methods will become widespread. A tool's ability to integrate with a computer-aided software engineering tool will become critical. Our synthetic-workload generator can easily become an integral part of a CASE tool.

Moreover, because SWSL is similar to current design notations, the generator can use high-level designs created by CASE tools to create synthetic workloads. As components of the application software are completed, they may be integrated into the synthetic-workload specification. Synthetic workloads can thus be used at all stages of system development, letting designers evaluate the effects of their decisions experimentally at each stage.

Since the synthetic workload will be used to evaluate embedded distributed real-time systems, we assume it will be compiled on a workstation and the executable code downloaded to the system. Because users cannot interact with the executing synthetic workload, they must specify any parameters that may change during execution beforehand and compile them into the workload.

SWSL defines an experiment as consisting of a number of statistically independent runs. During each run, the synthetic workload executes and the user collects performance data. A workload specification may contain parameter values for a number of runs. For each run, the user may define different values for the workload parameters, or use a single-value parameter for a set of consecutive runs. The generator compiles and downloads this single specification at the beginning of the experiment. The synthetic workload pauses between each run to let the user upload performance data or reset and adjust data-collection instruments. The user can specify values like the duration of each run and the random-number generators used to produce stochastic behavior in the synthetic workload.

To define synthetic workloads for a distributed system, SWSL specifies the processor on which each workload component resides. Components are statically allocated and can be easily moved from one processor to another between workload executions, so the user can change load distribution between experiments. SWSL also provides replicated objects. Multiple identical objects may be defined on multiple processors. Such objects may be used to represent objects that have been replicated for fault tolerance, or they may be representative members of a specific class of objects within the workload being modeled.

**Hmon.** Monitoring and debugging are topics of active research. Monitors are popular because they let users view the monitored system at various levels of complexity or abstraction. However, such monitors are typically intrusive and not applicable to real-time systems because of their unpredictable interference.

Monitoring distributed real-time systems is a challenge but one that must be met if designers, system architects, and performance evaluators are to measure, debug, test, and develop systems efficiently. Monitoring — the measurement, collection, and processing of information about task execution — can be complicated by system characteristics. A real-time system, for example, requires the monitor itself to operate under strict reliability and performance constraints. The reliability constraints require that both the monitored system and the monitor continue to operate in the presence of static or dynamic failures. The performance constraints require that the interference caused by the monitor's presence be predictable, minimal, and bounded.

Distribution also imposes constraints on the monitor. Distributed systems lack both global-state information and total ordering defined over events on different nodes. Users must have a way to collect monitored data from several sites and integrate it to get a coherent system view. Further, when tasks run in parallel, their behavior can be nondeterministic. The monitor must support deterministic task replay to enable effective debugging.

Passive hardware monitors can provide detailed, low-level information about a system, such as communication activities, memory accesses, and I/O patterns. They also cause little interference to the monitored system. However, hardware monitors do not support the interactive modification of task execution, which supports debugging.

To address these issues, we developed Hmon, a distributed monitor that runs on a dedicated processor.[8] Some system hardware is also dedicated to Hmon to minimize interference with the measured system. Figure 3 shows how data collection is monitored. Hmon provides transparent, continuous monitoring using dedicated hardware and integrated operating-system-level software. It runs on a dedicated application processor, called the monitor processor, on each node of HARTS. Additional code to collect data runs on the network processor and the application processors of each node.

Each processor's local memory is accessible to other processors. The monitored processors write data directly to the memory on the monitor processor. The monitor processor, in turn, logs the data on an external user workstation. Though the data-collection code interferes with the system being monitored, in our system, this interference is low, predictable, and accounted for in CPU and network scheduling. Since interference is the same during normal execution as it is during development and debugging, the debugging code is a predictable part of the application.

Hmon has three stages: extract data from the application processors and network processor, compress data on the monitoring processor, and log data on an external workstation. Data on monitored events is acquired through code inserted into the monitored system. We acquire much of our data by monitoring system calls transparently through modified pSOS and HARTOS call interfaces. Hmon also monitors interrupts, context switches, and shared variable references to allow deterministic task replay — a characteristic critical in debugging real-time programs. Without it, debugging commands destroy the timing-dependent nature of real-time systems. Moreover, breakpoints can be inserted only during replay, not during normal execution, or timing behavior becomes inconsistent.

The software monitor extracts data from the application processors, compresses data on the monitoring processor, and logs data on a workstation.

All extracted data is sent to the monitor processor. Each monitor processor orders and compresses the logged data and sends it from its node to an external user-level process running on a workstation outside HARTS. This process receives and archives data coming in from all monitor processors so that the events can be replayed for debugging. The monitor processors use the Ethernet controller on each node to send their data to the user

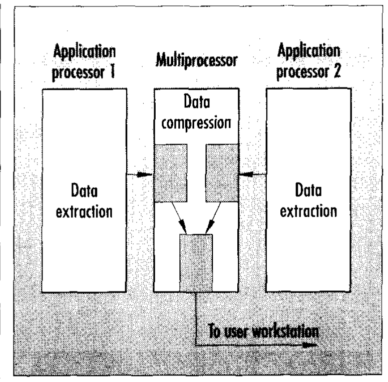


*Figure 3. Data-collection monitoring.*

workstation. The HARTS interconnection network remains unaffected by data transmission over the Ethernet.

We maintain predictability by keeping interference deterministic. Intrusive debugging is done only during replay, not on-line. Replay is done deterministically to make debugging feasible. Our approach is unique in that we perform transparent monitoring and deterministic replay without adding any special hardware such as a bus probe or a hardware-instruction counter.

Hmon supports services like debugging distributed real-time applications, aiding real-time task scheduling, and measuring performance. Because monitoring is transparent, user programs need no additional special code. The monitor is flexible enough to observe both high-level events that are system- and application-specific as well as low-level events like shared variable references.

**Fault injector.** The fault injector inserts errors into an otherwise error-free system. The user controls the error type and location. The injector lets the user evaluate dependability mechanisms on HARTS. Real-time systems used in life- and mission-critical tasks have many fault-tolerance and

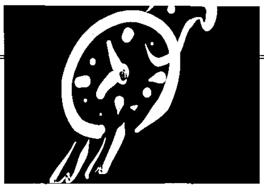


**TABLE 1**
**COMMUNICATION TIMES FOR REMOTE CALLS**

| Call | Local time (ms) | HARTOS times (ms) | | |
|---|---|---|---|---|
| | | Intranode | Internode | Difference |
| rsend_x | 0.104 | 2.799 | 4.615 | 1.816 |
| rjam_x | 0.103 | 2.799 | 4.614 | 1.815 |
| rliber_x | 0.113 | 2.949 | 4.786 | 1.837 |
| rresume_p | 0.079 | 2.333 | 4.053 | 1.720 |

**TABLE 2**
**DATA-TRANSFER TIMES**

| Message size (bytes) | Time (ms) | Message size (Kbytes) | Time (ms) |
|---|---|---|---|
| 256 | 5.033 | 2 | 12.590 |
| 512 | 5.730 | 4 | 20.540 |
| 768 | 6.380 | 8 | 33.880 |
| 1024 | 6.998 | 12 | 47.118 |
| | | 16 | 60.490 |

fault-recovery mechanisms to ensure high dependability. These mechanisms must be rigorously tested to verify that the system meets its dependability goals. Such testing can be very difficult, given the large mean time between failure of highly dependable systems. To verify the properties of such a system experimentally, there must be some way to accelerate the occurrence of faults or errors. A fault injector lets the user introduce faults or errors into the system.

Most fault-injection experiments fall into one of two categories: hardware-fault injection and software-fault injection. In hardware-fault injection, faults are typically inserted into the system at the pin level. In software-fault injection, errors are typically inserted by altering the contents of memory or registers. Errors can also be introduced by corrupting messages or altering object code.

The fault injector lets you inject transient, intermittent, and hard faults at each node.

The fault injector in HARTS supports a variety of faults and errors, each of which can be injected as transient, intermittent, or hard faults. In addition, the injector lets different faults be injected at each node in the system. The injection time and duration can also be specified. This can greatly simplify the testing of dependable distributed applications, which often must be able to tolerate erroneous behavior by multiple nodes in the system over an extended period of time. In addition, the injector is easy to integrate with any workload developed for HARTS.

The injector provides a tool suite to simplify and automate the design and execution of dependability experiments. Its two main components are the experiment generator and the control modules. The experiment generator creates the executable and script files to run the fault-injection experiments from a user-supplied experiment-description file. The control modules consist of the routines that provide the actual fault-injection capability. The experiment generator compiles the appropriate portions of the control modules with the workload for each node.

To create and run an experiment using the injector, you create an experiment-description file that provides the injector the names of the HARTS nodes to be used, the location of the workload, and the type of faults to be injected on each node. The workload can be a real application or a synthetic workload generated by the synthetic-workload generator described earlier. Once the experiment description file is created, you run the experiment generator to create all the executable and script files needed to run the fault-injection experiment. During the experiment, you can use Hmon to collect performance data.

The fault types used by the injector are memory faults, communication faults, and processor faults. The memory faults are injected as single-bit or burst errors. Communication faults cause lost, altered, or delayed messages. Processor faults represent faults in the CPU's functional units.

Each fault type has many possible variations, which the user can specify. The injection of memory and communication faults can be transient, intermittent, or permanent. If the fault is to occur intermittently, the user can specify a probability distribution that describes the interarrival times between faults. For memory faults, the user can emulate permanent faults by specifying small interarrival times for an intermittent fault. This emulation is not the same as a true permanent fault, however, because the workload can overwrite the faulty location between injections.

By using combinations of these faults on each node, the user can implement a variety of failure semantics.

## STATUS

Version 1 of HARTOS is completed and has been stable for more than two years. We used it to provide interprocessor communication for the development of the synthetic-workload generator and Hmon. We are not planning to develop it further.

We timed several classes of operations for HARTOS version 1 using the baseline measurement of a single task sending a message to a single task. All measurements were done on an Ethernet, which was otherwise idle.

Table 1 shows the first set of measurements, which are the communication times for remote calls. We timed a variation of the call that sends a short message to an exchange and compared it with the time required to send a resume signal to a task. Local time (second column in the table) is the time required to execute the corresponding pSOS calls on a single processor. Intranode refers to operations between two processor cards in the same node; Internode refers to operations between processors in different nodes connected by the Ethernet. All intranode and internode calls were made in blocking mode.

The intranode communication using the network processor shows the overhead involved in assembling and interpreting a message and maintaining the connection structure in the network processor. Difference is the difference between the intranode and internode times — the time required to transmit and receive two messages (request and reply) on the Ethernet and the cost of setting up a packet timeout. In addition to the actual network-transmission time, values under Difference include the cost of initiating packet transmission and setting up receiving buffers with the Ethernet controller. For small packets, the Ethernet controller's processing time is actually longer than the network-transmission time.

Table 2 shows the second set of measurements, which are for data-transfer operations. These values represent the time required to transfer $x$ bytes of data across the network to another process. The data-transfer operations show a close-to-linear increase in communication time with an increase in message size. This relationship holds over two ranges of message sizes: less than 1 Kbyte and 2 to 16 Kbytes. There is a small jump in the communication time for sizes greater than 1 Kbyte because they require a multipacket message, which has two acknowledgment packets. However, the communication cost per byte is less for larger messages because not all packets require acknowledgments.

Version 2 of HARTOS is under development. It currently provides all the functionality of version 1. The clock-synchronization protocol has not yet been implemented because it depends on the programmable routing controller, which is under development. However, we have implemented many of the functions of the real-time-channel service and measured their performance.

We have also performed some preliminary measurements to estimate the time to establish a channel in a 19-node hexagonal mesh. For a channel with three links — the longest possible in a mesh that size — we estimate that it will take 43.5 ms to establish a channel. We cannot make any other measurements without the custom network-processor architecture, which is still being developed.

Much work remains on version 2 of the HARTS operating system and on the evaluation tools. The synthetic-workload generator is operational and has been used for the performance measurements obtained for HARTOS version 1. As HARTS and HARTOS develop, we will use the generator extensively to evaluate them. A range of experiments have been planned. We plan to first create simple synthetic workloads to measure basic performance values and test functionality. We will then use the generator to produce synthetic workloads that are representative of sample applications. We will use these to measure HARTS performance under the load offered by each sample application. We plan to study the performance of the HARTOS network-communication facilities, the effects of asynchronous tasks on the ability to schedule tasks, message scheduling, and the performance of fault-tolerance mechanisms.

The debugging features of the monitor are incomplete and will be expanded. We plan to improve the debugger interface and further develop the monitor processor to enhance deterministic replays and support CPU scheduling. We will also explore a utility to analyze monitored data on a workstation.

The fault injector is partially complete. The experiment generator has been implemented, as has the injection of most of the fault types. We are still implementing the rest of the fault types and improving the data-collection and -analysis tools. The fault injector is intended as a support tool for dependability experiments on HARTS. As a result, it will be continuously expanded as new applications arise that require different capabilities. ♦

## ACKNOWLEDGMENTS

The work reported here is supported in part by the Office of Naval Research under contract N00014-92-J-1080 and grant N00014-91-J-1115, and the National Science Foundation under grants MIP-9012549 and MIP-9203895. Any opinions, findings, and conclusions or recommendations expressed in this article are ours and do not necessarily reflect the views of the funding agencies.

## REFERENCES

1. N. Hutchinson and L. Peterson, "The $x$-Kernel: An Architecture for Implementing Network Protocols," *IEEE Trans. Software Engineering*, Jan. 1991, pp. 1-13.
2. P. Ramanathan, D. Kandlur, and K. Shin, "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems," *IEEE Trans. Computers*, Apr. 1990, pp. 514-524.
3. D. Kandlur and K. Shin, "Design of a Communication Subsystem for HARTS," Tech. Report CSE-TR-109-91, CSE Division, Department of EECS, University of Michigan, Ann Arbor, 1991.
4. P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, Sept. 1979, pp. 267-286.
5. D. Kandlur and K. Shin, "Reliable Broadcast Algorithms for HARTS," *ACM Trans. Computer Systems*, Nov. 1991, pp. 374-398.
6. A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, Feb. 1984, pp. 39-59.
7. D. Kiskis and K. Shin, "A Synthetic Workload for Real-Time Systems," *Proc. Workshop on Real-Time Operating Systems and Software*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 77-81.
8. P. Dodd and C. Ravishankar, "Monitoring and Debugging Distributed Real-Time Programs," *Software Practice and Experience*, to appear.

**Kang G. Shin** is professor and chair of the computer science and engineering division of the electrical engineering and computer science department at the University of Michigan. His interests are distributed real-time computing and control, fault-tolerant computing, computer architecture, and robotics and automation. He has written or coauthored more than 190 papers and several book chapters in these areas. He is also an editor for *IEEE Transactions on Parallel and Distributed Computing* and *International Journal of Time-Critical Computing Systems*.

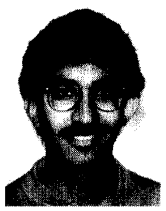
Shin holds a BS in electronics engineering from Seoul National University, Korea, and an MS and a PhD in electrical engineering from Cornell University. He is an IEEE fellow, a distinguished visitor of the IEEE Computer Society, and chairman of the IEEE Technical Committee on Real-Time Systems.

**Dilip D. Kandlur** is a research staff member at the IBM T.J. Watson Research Center, where his interests are operating systems, real-time systems, and networks. He participated in the work described in this article while at the University of Michigan.

Kandlur holds a BTech from the Indian Institute of Technology, Bombay, and an MSE and a PhD from the University of Michigan — both in computer science and engineering. He is a member of the IEEE Computer Society.

**Daniel L. Kiskis** is a PhD candidate in computer science and engineering at the University of Michigan. His research interests include synthetic workloads, real-time workload characterization, and performance evaluation.

Kiskis holds a BS in computer science and mathematics from Denison University and an MSE in computer science and engineering from the University of Michigan. He is a member of the IEEE, Sigma Xi, and Phi Beta Kappa.



**Paul S. Dodd** is manager of research and development at Myra Systems Corp., where his interests include real-time monitoring, operating systems, and networks.

Dodd holds a BSE in computer engineering an MSE in computer science and engineering, both from the University of Michigan, where he participated in the work described in this article.

**Atri Indiresan** is a PhD candidate in computer science and engineering at the University of Michigan and a research assistant at the university's Real-Time Computing Laboratory. His research interests include operating system and architectural support for performance guarantees and fault tolerance.

Indiresan holds a BTech in computer science and engineering from the Indian Institute of Technology, Madras. He is a member of the IEEE Computer Society and Tau Beta Pi.

**Harold Rosenberg** is a PhD candidate in computer science and engineering at the University of Michigan and a research assistant at the university's Real-Time Computing Laboratory. His research interests include software fault injection, fault tolerance and real-time operating systems.

Rosenberg holds a BS in electrical engineering from Tufts University and an MS in electrical and computer engineering from the University of Massachusetts at Amherst. He is a member of the IEEE, ACM, Tau Beta Pi, and Eta Kappa Nu.

Address questions about this article to Shin at Real-Time Computing Laboratory, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122; Internet kgshin@alps.eecs.umich.edu.