HARTOS: A Distributed Real–Time Operating System

Dilip D. Kandlur, Daniel L. Kiskis, and Kang G. Shin

Real–Time Computing Laboratory
Department of Electrical Engineering & Computer Science
The University of Michigan
Ann Arbor, MI 48109–2122.

e-mail: kgshin@zippy.eecs.umich.edu

Abstract

This paper outlines the design objectives and research goals for HARTOS, a distributed real-time operating system being developed at The University of Michigan. This effort is part of a larger research project to design and implement an experimental distributed real-time system called the Hexagonal Architecture for Real-Time Systems (HARTS). An important feature of HARTS is the use of an intelligent network processor to handle many of the functions relating to communications. The paper focuses on the communications aspects of the operating system and the control software kernel of the network processor. The preliminary version of the kernel provides good support for inter-process communication and distributed control. Its performance has been measured and analyzed and found to be comparable to that of other message passing systems like the V system.

1 Introduction

The availability of inexpensive and powerful microprocessors has led to the use of multi-computers in an increasing number of critical real-time applications. A multi-computer system is suitable for environments which have considerable physical separation between the components to be controlled. Many of these applications require their computers to provide fast and reliable operation. At the Real-time Computing Laboratory (RTCL), The University of Michigan, we are investigating the specification, design and implementation of distributed real-time systems. To study the low-level architectural issues, we are currently building an experimental distributed real-time system called the Hexagonal Architecture for Real-Time Systems (HARTS). This paper deals with the operating system for HARTS, called HARTOS.

HARTS is comprised of several multi-processor nodes connected by a *hexagonal mesh* interconnection network [1, 2]. This is a regular, homogeneous network in which each node has six neighbors and is constructed by systematically adding wrap links to a planar hexagonal graph. In addition to

The work reported in this paper was supported in part by the Office of Naval Research under Contracts N00014–85–K–0122 and N00014–85–K–0531 and by an IBM Graduate Fellowship.

supporting efficient algorithms for routing and broadcasting, the hexagonal mesh also has some interesting topological properties [1]. The number of nodes in a wrapped hexagonal mesh is related to its diameter d by the formula: $3d^2 + 3d + 1$. The version of HARTS under construction has a hexagonal mesh of diameter 2 and contains 19 nodes. Although point-to-point interconnection networks like the hexagonal mesh provide good fault-tolerance properties, they can potentially have a higher latency for message delivery compared to a broadcast network when packet switching is employed. To reduce this latency, HARTS uses a *virtual cut-through* switching scheme [3] which has latencies similar to circuit switching under light to moderate loading. In this scheme messages arriving at an intermediate node do not always get buffered, instead, they are forwarded to the next node in the route if a circuit can be established.

Contemporary research in real-time operating systems has been focused mainly on processor scheduling and synchronization. Issues relating to inter-process communication have been addressed mainly for multi-processor systems. Network communications, which forms an integral part of a distributed real-time system, has received relatively less attention. Our operating system efforts have mainly been directed towards the communications subsystem with the objective of providing efficient and fault-tolerant communications. The use of a communication processor to handle many of the functions relating to communication is an important aspect of this work. Our objective is also to develop a vehicle for experimenting with communications protocols, message scheduling schemes and low-level routing algorithms.

In this paper, Section 2 discusses the design objectives and research issues of the system. Section 3 describes the HARTS system while Section 4 gives an overview of the current version of HARTOS. In Section 5 we describe the operations of the communications subsystem. The performance of our current version is analyzed in Section 6. Conclusions and future directions are presented in Section 7.

2 Design Objectives

Although HARTOS is primarily targeted for HARTS, only the low-level network operations are dependent on that structure. For the high-level design, we assume an architecture consisting of multi-processor nodes connected by a general, fault-tolerant network. The nodes are bus-based and contain one or more application processors (APs), supported by a network processor (NP). Application processors run copies of the distributed kernel while the network processor handles most of the kernel functions relating to communication.

The communications requirements of real-time processes range from reliable "stream" communication to periodic sampling in which loss of samples can be tolerated. One of our primary design goals is to support these diverse requirements through a small set of mechanisms using parameterized functions. The kernel aims to provide a uniform interface for communication between processes independent of their location. It handles three types of communicating processes: processes within an AP, processes on different APs connected via a common bus within the same node, and processes on APs on different network nodes. However, it also allows applications to make use of shared memory in the case of processes on the same multi-processor node. There are two mechanisms provided for inter-process communication: a message send/receive facility and non-queued event signals. These can also be used for synchronization between processes using special messages. Other interesting research issues in this design are:

- **Support for replicated processes:** The mechanism for fault-tolerance envisaged in our system is replication of processes. This is a process level version of N–Modular Redundancy (NMR) and voting schemes like those used in many fault-tolerant systems such as FTMP [4]. To support multiple versions of a process, a *process group* mechanism will be used. The replicas of a process would belong to the same group and operations which are normally available for processes would be available for process groups, including event signalling, process control, etc. The message passing scheme is based on mailboxes and *distribution lists* are associated with mailboxes. Messages sent to these lists would be delivered to the member mailboxes and they would in turn be received by all the processes using them. Thus, messages could be sent to all copies of a process.

- **The Network Processor (NP):** The NP would handle many of the operating system functions relating to communication for the multi-processor node. It would also handle functions like system monitoring and would implement algorithms for broadcast and multicast messages to support the process group mechanism. The intent is to minimize the overheads experienced by APs due to inter-processor communications so that the computing power available to user processes is not degraded. However, an increased load on the NP could adversely affect the throughput of the network. We intend to study this trade-off quantitatively by experimenting with the design.

- **Real-time Communication:** We are currently analyzing scheduling schemes for packets with deadlines on the delivery time in a multi-hop network like the hexagonal mesh. In particular, we have examined the efficacy of employing local deadlines for each hop in the route as opposed to enforcing a single deadline for end-to-end delivery. An important consideration for message scheduling in HARTS is efficient use of virtual cut-through capability. The objective here is to schedule messages and select routes such that the number of store-forward hops for the packet is minimized.

- **Fault-tolerant Routing:** The hexagonal mesh interconnection network possesses the ability to withstand multiple node and link failures. It also presents challenges for the design of algorithms for routing and broadcast which are tolerant to failures. In [5], we have developed a routing scheme for HARTS, which assures the delivery of every message as long as the destination is reachable. The scheme can also detect the non-existence of a path between a pair of nodes in a finite amount of time. Moreover, the scheme requires each node to know only the state (faulty or not) of each of its own links. The most impressive finding during the simulation of this scheme for a HARTS of diameter 2 is its survivability: even when more than 50% of links have failed, approximately 95% of messages are deliverable by this scheme. This scheme could be incorporated into the low-level subsystems of the kernel relating to message delivery.

While work on some of these issues is currently underway, we have completed a preliminary version of HARTOS. The issues handled in this version relate to inter-process communication and use of the network processor.

3 Operating Environment

RTCL is currently equipped with several VME-bus based Ironics Performer-32 systems. Each system has 1–3 processor cards and an Ethernet processor card. The processor cards have a Motorola 68020 32-bit processor, optional memory management unit, and 1 or 4 Mbytes of dual ported RAM which can be accessed from the VME-bus. The Ethernet Processor card ENP-10 uses a 10 MHz 68010 processor, an AMD Ethernet Controller device (LANCE), and provides 512 Kbytes of buffer memory which is also accessible from the VME-bus. The processor cards also have a hardware mailbox interrupt mechanism which generates a CPU interrupt on a write access to the top 256 bytes of the dual-port memory. These multi-processor nodes are currently connected only through the Ethernet since the hexagonal mesh network is under development. The Ethernet also serves as a link to the workstations used for software development. A custom routing controller chip [2] has been developed which supports virtual cut-through switching. It implements the data link layer and portions of the network layer of the OSI seven layer model for network communications. This chip will be used in the network processor card and will serve as the front-end interface to the hexagonal mesh.

4 Preliminary Version of HARTOS

The preliminary version of HARTOS utilizes the pSOS™ [6] uniprocessor real-time operating system kernel. The motivation for building this version was to provide a facility to construct distributed real-time applications and studies related to workload characteristics. pSOS serves as the executive on each processor and provides facilities like process management, memory management, event handling and inter-process communication. Table 1 gives a summary of the kernel calls. The HARTOS kernel extends many of these facilities to a *multi-processor, multi-computer* environment and essentially provides the required communication support. This section presents a brief overview of pSOS followed by a description of HARTOS functions.

The process management facilities provided include process creation, suspend, resume, change mode, and set priority. A *process* is the unit for sequential execution, resource ownership and scheduling. Processes are created dynamically and can be grouped together into "process groups" which serve as a unit for protection for several operations including inter-process communication. The base kernel does not however support group control operations. The kernel operates in a non-mapped environment and processes on an AP share a single linear address space. Processes have names associated with them which can be used by other processes to obtain the internal process_ids using the available mapping functions. The kernel employs a pre-emptive priority based scheduler with a provision for cyclic service for processes of equal priority. Processes can dynamically change their priority and also protect themselves from interruption during critical sections by setting their mode to non-preemptive.

The primary mechanism used for inter-process communication is the *message exchange*. A message exchange is an object at which messages or processes can be queued. The exchange thus allows many-many process communication. A message exchange is identified by its exchange_id but it also has a name associated with it. Processes which want to send or receive messages from the exchange can obtain the exchange_id by presenting the name of the exchange using the attach_x function. Messages, which are small six-longword records with a 2-word header and a data region of four long-words, can be sent to an exchange from one or more processes, and one or more processes can

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

Process Management	
SPAWN_P(name, grid, prior, stksize, arglist, pid)	Spawn a new process
ACTIVATE_P(pid, start_adr)	Activate a new process
IDENT_P(name, pid, status)	Get the id and status of a process
DELETE_P(pid)	Delete a process
SUSPEND_P(pid)	Suspend a process
RESUME_P(pid)	Resume a suspended process
PRIORITY_P(pid, increment, old_prior)	Change a process's priority
MODE_P(new_mode, mask, old_mode)	Change caller's mode bits
SUPER_P(old_stat_reg)	Put caller into supervisor state
Memory management	
ALLOC_SEG(size, region, wait, timeout, segment)	Allocate a segment from a region
FREE_SEG(segment)	Return a segment to a region
ASSIGN_SEG(segment, to_pid, newadr)	Pass ownership of a segment
GRAB_SEG(segment, from_pid, newadr)	Take ownership of a segment
CREATE_PT(memblock, buf_size, n_buf, ptid)	Create a partition of buffers
DELETE_PT(ptid, buf_size)	Delete a partition of buffers
GET_BUF(ptid, buf_size, buf_adr)	Get a buffer from a partition
RET_BUF(ptid, buf_adr)	Return a buffer to a partition
Communication and synchronization	
CREATE_X(name, property, msgQ_max, xid)	Create a message exchange
ATTACH_X(name, xid, proc_Q, msg_Q)	Get exchange_id and status
DELETE_X(xid)	Delete a message exchange
REQ_X(xid, message, wait, timeout_option)	Get or wait for message from exchange
SEND_X(xid, message)	Post message at end of an exchange queue
JAM_X(xid, message)	Put message at head of an exchange queue
LIBER_X(xid, message, min_cnt, nmsg, nlib)	Broadcast a message to an exchange
WAIT_V(event, condition, timeout, caught_v)	Wait for events
GET_V(event, condition, caught_v)	Get or reset events
SIGNAL_V(pid, events)	Signal events to a process
Time management	
ANNOUNCE_T()	Inform pSOS of a clock tick
SET_T(date, time, ticks)	Set time and date
GET_T(date, time, ticks)	Get time and date
PAUSE_P(ticks)	Delay a process by an interval

Table 1: pSOS system calls
(summarized from [6])

receive messages from the same exchange. The message exchange mechanism is also used for process synchronization where send and receive to an exchange are analogous to the P and V operations on a semaphore. The kernel recognizes certain special messages, which have a "home-exchange" specified in the message header, and keeps an internal copy of these messages. This copy is used to return any special messages owned by a process when the process is deleted. The *signal* facility is an alternate low-cost mechanism for inter-process communication by which *events* can be posted to a process.

The kernel uses an I/O interface based on a device switch table similar to that used in the UNIX™ System. Although device drivers are application specific, good support is provided for fast I/O. Interrupt handlers are treated as special processes and they can use a restricted set of primitives to interact with other processes.

4.1 Distributed Kernel Functions

The functions supported by HARTOS can be classified into several groups: process control, inter-process communication, time queries, and name lookup. A summary of these functions in given in Table 2.

Most of the functions are parameterized and a user process has control over several parameters in their execution like the number of retries, and whether the call is blocking or non-blocking. A blocking call is one in which the user process waits until the operation is completed on the remote processor and the results are returned. For a blocking call, a process can specify a timeout period which is the maximum duration that it is willing to wait. This leads to several interesting variations with differing overheads and degree of reliability.

- A no retry non-blocking mode that corresponds to a unreliable datagram service.
- A non-blocking mode with retries which gives a responsive-send.
- A blocking mode with retries which is the standard RPC mechanism.

While using calls with a non-blocking mode there is a potential for multiple incomplete calls in the system. The standard mechanism of queueing these operations tends to create problems like flow control. The approach taken here is to discard the earlier operation when a new operation arrives for the same destination address. This is consistent with the requirements of many real-time applications,

™UNIX is a trademark of AT&T Bell Laboratories.

Name Management	
rENTER_S(name, class, obj_id)	Enter a name into the name table
rFIND_S(name, class, mode, retries, timeout, obj_id, cpuid)	Find a name
rREMOVE_S(name, class)	Remove a name from the name table
Process Management	
rRESUME_P(cpuid, pid, mode, retries, timeout)	Resume a suspended process
Communication and synchronization	
rSEND_X(cpuid, xid, msg, mode, retries, timeout)	Post message at end of an exchange queue
rJAM_X(cpuid, xid, msg, mode, retries, timeout)	Put message at head of an exchange queue
rLIBER_X(cpuid, xid, msg, min_cnt, mode, retries, timeout, nmsg, nlib)	Broadcast a message to an exchange
rSIGNAL_V(cpuid, pid, events, mode, retries, timeout)	Signal events to a process
Time management	
rANNOUNCE_T(cpuid, mode, retries, timeout)	Inform pSOS of a clock tick
rSET_T(cpuid, date, time, ticks, mode, retries, timeout)	Set time and date
rGET_T(cpuid, date, time, ticks, mode, retries, timeout)	Get time and date
Data transfer	
rDATA_SEND(cpuid, pid, msg, msg_length, mode, retries, timeout)	Send a message to a remote process
rDATA_RECV(msg, msg_length, to_loc)	Receive a data transfer
rWAIT_C()	Wait for data transfer to complete

Table 2: HARTOS system calls

like control applications in which sensor data is gathered periodically. The *rwait_c* function is also available to allow a process to block until completion of an outstanding operation.

The inter-process communication primitives related to message exchanges are *rsend_x*, *rjam_x*, and *rliber_x* which post short pSOS messages to exchanges. *rsend_x* and *rjam_x* differ only in the way in which the message is placed on the exchange queue, which is the tail or the head of the queue, respectively. The *rliber_x* posts multiple copies of the message to the exchange and can be used to unblock all processes that are blocked at the exchange. The *rsignal_v* function is used to signal events to processes. The time related functions are *rset_t*, *rget_t*, and *rannounce_t* which allow a process to access the time on remote nodes.

The short message length used with the exchange mechanism is adequate for processes on a single processor or on a multi-processor with shared memory as large messages can be passed using buffers in the shared memory. In the distributed kernel however, an alternate mechanism for large data transfers is needed. For large messages we have the *rdata_send* and *rdata_recv* operations which transfer up to 16 Kbytes of data between processes. These operations also take a *priority* parameter which is used for placing an order on operations from different processes. Although it is conceivable to use priorities for other operations, this is not required since the resource and time requirements of those operations are small.

Each of these calls uses an internal address which consists of an *AP_ID* and an exchange or process ID to specify the destination. A naming service is provided to locate objects and a process can obtain these internal addresses using the name lookup mechanism. Objects can be registered into the name service using the *renter_s* function which establishes a name to address correspondence. The *rfind_s* function does a name lookup and returns the stored ID. While the service is used primarily to locate communication objects, it can also be used to register and locate other objects.

5 The Communications Subsystem

An important feature in HARTS is the use of the intelligent network processor. This handles all network transmissions, receives packets, interprets them and presents them to the appropriate AP. As the hexagonal mesh interconnection network and the custom network processor are being developed, we have used the Ethernet as the interconnection network and the ENP-10 Ethernet processor as the network processor. It is, however, expected that the hexagonal mesh network processor would have a similar architecture from the software viewpoint.

The distributed kernel functions are handled using a variation of the remote procedure call (RPC) mechanism [7]. The RPC binding is explicit since the destination address is specified in the function call. At the logical level, the operations involved in a remote function call can be described as follows. The processor AP1 forwards the data to the NP. The NP marshalls the data into a network packet and transmits the packet to the NP (NP2) on the destination node. NP2 now interprets the message, determines the destination AP and forwards the message to that AP (say AP2). AP2 executes the function and returns the reply to NP2. The reply then traces its way back to AP1. The server side handling for these functions is non-blocking and very short, and can be executed as part of the interrupt handler. This eliminates the need for kernel server processes which are used in many RPC systems. This sequence also applies to function calls within the same multi-processor node but without the network transmissions. This mechanism suffices for all functions except the *rdata_send*. In this case, the message is queued for the process in the NP and is retrieved using an explicit *rdata_recv* call by the receiving process.

The communications subsystem assumes that the underlying transport is an unreliable datagram service and builds a reliable sequenced transport on top of it. Each function call is a request–reply transaction and is identified by a transaction sequence number. These sequence numbers are specific to a particular [AP, process] pair and are monotone increasing. A function call is completed when the reply packet is received. Thus a reply packet acts as an acknowledgement for the request. In case a reply is not received within a certain time period, the request is retransmitted with the same sequence number. The receiver side uses the sequence numbers to eliminate duplicate requests and to discard out-of-sequence arrivals. It also holds on to reply packets for a certain period in anticipation of lost replies. For non-blocking operations in which retries are not required, the reply packet is not necessary and the request packet is treated like an unreliable datagram. We note that on a single Ethernet, there is no possibility for out-of-sequence arrivals. However, the hexagonal mesh is like a collection of networks and intermediate nodes are like gateways, and there are multiple paths between nodes. So, congestion could cause packets to arrive out-of-sequence.

The *rdata_send* operation is the only one which can possibly require a multi-packet transmission. Since the error rates on local networks are low, we do not need a per packet acknowledgement for the packets in the message [8]. The scheme employed is a burst mode transfer with group acknowledgements. The sender and receiver use a bit-vector representation to keep track of packets within the message. The sender can request an acknowledgement on any packet, and the receiver returns a cumulative acknowledgement for all the packets that it has received. A link-level flow control mechanism is employed to ensure that there is no packet assembly deadlock in the system [9]. The first packet in a message is treated as a buffer reservation request and the destination responds with a positive ACK only if the request can be honored. Thus under ideal operating conditions, there are only two ACKs needed for the message (one if the message consists of a single packet). On the receive side, the messages are queued up for the process to which they are addressed. They are retrieved from the NP by a *rdata_recv* operation. Here, the receiving process presents a buffer into which the message is to be copied. If there is a message queued, the NP copies it into the buffer. Otherwise, if the process specifies the block option, the request is held pending arrival of a message. The communicating processes are assumed to be responsible for user-level flow control and so there is no partitioning imposed on the buffers in the NP.

In addition to the functions described above, the HARTS NP will have some additional functions specific to the hexagonal mesh network. The NP must handle packets which are in transit but have been buffered at the NP due to an inability to establish a circuit to the next node in the route. Also, since there can be multiple possible paths to the destination, the NP has to select an appropriate route. The frequency of this buffering of packets is however, expected to be small. Simulation and analytical studies for specific workload patterns [10] have shown that under light to moderate loading conditions, over 90% of the packets get delivered without the need for buffering at intermediate nodes.

5.1 Name Service

The NPs maintain a distributed Name Table which is used to map logical names to internal addresses. Each NP maintains a table of map entries for entities that were declared on that node, that is, by the associated APs, and entries to this table can be made using the *renter_s* function. A process may locate a named server by submitting an *rfind_s* request to the NP. On a *rfind_s* operation, the local name table is first searched for a match. In case there is no local match, a request for a name mapping is broadcast to other NPs on behalf of the process that made the request. Only NPs which detect a match reply to the request and the first reply received is taken. The sequence of actions is similar to that of a remote call except that the request is handled entirely by the NPs and is not visible to the APs.

5.2 Implementation Details

The communication system for HARTOS consists of agents on both the AP and the NP. On the AP side we have the reentrant stub interface routines for the calls, a common network agent to communicate with the NP, and an Interrupt handler. Communication between the APs and the NP is through mailboxes in the memory of each AP. Two mailboxes are used on each AP. One is for passing data to the NP for a remote call. The reentrant stub interface routines extract some of the call parameters and place them into CPU registers. They then trap to the network agent which synchronizes access to the NP and places the request into the mailbox. If the function specifies a blocking operation mode, the process is then suspended awaiting a completion signal from the NP. Once the parameters of the call are placed in the mailbox, a flag is set in the memory of the NP. Access conflicts between processes on the AP are prevented by using a non-preempt mode while accessing the mailbox. The other mailbox is used to pass data from the NP to the AP. This data can be either the parameters of a remote call or a reply from a call which originated from the AP. When data is placed in this mailbox, the Interrupt handler is invoked to execute the remote call or to transfer the results to the AP. Access to the mailboxes is governed by a simple producer-consumer type protocol with a buffer size of one.

The NP program consists of several logical processes corresponding to handlers for different operations. There are separate handlers for message send, packet receive, timeout and data transfer. These are scheduled by a control loop which polls the flags associated with the AP mailboxes and such structures as the queue of incoming packets. When one of these data structures indicates that a task is to be performed, the appropriate handler is activated. The handlers run to completion and there is no processing done during interrupts. Thus the number of possible states is reduced and access to common structures is simplified.

The interface between the NP program and the Ethernet Controller (LANCE) is through the K1 kernel [11]. The K1 kernel controls the sending and receiving of messages. Outgoing messages are submitted to the K1 kernel by the NP program. The K1 kernel queues the messages for transmission by the LANCE. It also provides an interrupt service routine (ISR) to pass received packets to the NP program. When a packet arrives for the NP, the K1 kernel receive ISR calls the NP program's receive ISR which places the packet on a queue of newly received packets. The status of this queue is checked during the NP program polling sequence. When a packet is present on the queue, the receive handler is called.

The K1 kernel also provides a timeout handling facility which is used by the NP program. The NP program sets timeouts for outgoing messages and transactions if requested by the application process. When a timeout occurs, the K1 kernel activates the timer ISR which queues the the timeout control block on the timeout queue. This queue is also checked during the NP program polling sequence and the timeout handler is activated if a timeout has occurred.

The NP uses buffers of two sizes: a short 100 byte buffer for requests, replies and ACKs and a long 1024 byte buffer for data transfer operations. Data for most remote calls and replies are copied from the AP mailbox into a short buffer which is submitted directly to the K1 kernel for transmission. The Datasend handler uses long buffers and sends out one packet at a time so that it does not block out the other handlers during a long message transmission. All packets are received from the network into long buffers.

6 Performance Measurements

We have measured timings for several different classes of operations on the HARTOS kernel. Measurements of individual kernel operations were performed by repeating the operation a large number of times and recording the elapsed clock time. The clock used was the software clock maintained by the local node kernel which was set to a resolution of 5 milliseconds. The measurements were conducted on an Ethernet that was otherwise idle.

The first set of measurements (see Table 3) gives a summary of communication calls for different types of communicating processes. The "local time" values are the times required to execute the corresponding pSOS calls on a single processor. The "intranode" category refers to operations between two processor cards in the same node while "internode" refers to operations between processors in different nodes connected by the Ethernet. The internode and intranode calls were blocking and a timeout of 3000 msec. was specified.

The second set of measurements were for the data transfer operations (see Table 4). These values represent the time required to transfer the given number of bytes of data across the network to another process. In each case, the destination process had executed a *rdata_recv* call and was thus waiting for the data at the time of its arrival.

Operations in the last category are local kernel operations and other parameters of interest like context-switching time and the Interrupt handler overhead. These have been summarized in Table 5. A more fine-grained measurement of the NP program was also performed to determine the time spent for the various operations, including the interactions with the APs using an intranode *rsignal_v* operation. These revealed that the AP overhead is approximately 284 microseconds on the send side and 123 microseconds on the receive side (see Table 5). The costs on the send side include the cost of making a request to the NP and the cost of receiving results from it. While making a request, the AP side routines place the parameters into the send mailbox and make a *wait_v* call to wait for the

Call	local Time	HARTOS Intranode Time	Internode Time	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 3: Categories of remote communication calls (time in msec.).

Message Size (bytes)	Time (msecs)	Message Size (Kbytes)	Time (msecs)
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

Table 4: Data transfer times.

reply. On the return path, there is an Interrupt signalling arrival of the reply, a transfer of the results to the process, and a *signal_v* call to wake up the blocked process. The costs on the receive side are essentially the cost for the Interrupt and the cost of performing the requested operation.

The timings reported in Table 3 are for a blocking operation with a message timeout specified. The time taken for an intranode rsend_x call was 2.336 msec. when no message timeout was specified and 2.799 msec. when a message timeout was specified. This reveals the overhead for setting and resetting a timeout, which is 463 microseconds. The times for internode non-blocking calls which

Operation Type	Time (microseconds)
Context-switch	95
Interrupt Response	31
Dummy rwait_c call	625
Local rfind_s call	750
Sender AP Time (rsignal_v)	284
Destination AP Time (rsignal_v)	123

Table 5: Miscellaneous operations.

did not require a reply message were approximately half the time for the corresponding blocking operation. The difference between the internode and intranode time is essentially 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 this includes the cost of initiating the packet transmission and setting up receive buffers with the Ethernet controller (LANCE). For small packets, this cost of initiating transmission and reception of packets with the K1 kernel overshadows the network transmission time.

The intranode operations also show the overheads involved in assembling and interpreting a message and maintaining the "connection" structures in the NP. Although this approach is useful for analyzing the NP program, intranode operations could be implemented better by direct communication using shared memory between the source and destination APs. It is expected that the effort required to implement these changes will not be substantial.

The data transfer operations (Table 4) show a close to linear increase in communication time with an increase in the message size. This holds over two ranges of message size: less than 1K and 2K to 16K bytes. There is a small jump in the communication time for sizes over 1K because in that case a multi-packet message is necessary which has two ACK packets. However, the communication cost per byte is less for larger messages because ACKs are not required for all packets.

It is observed that the time required for remote operations is higher than similar measurements reported for the V kernel [12]. The time reported there for a Send-Receive-Reply operation was 2.54 msec. with a 10MHz processor. The V kernel measurements are relevant since the hardware for which they were reported is similar to the ENP-10 card used here. The longer time reflects the additional overheads involved in copying data across the VME-bus between the AP mailboxes and the NP where the network packet is formed. However, although the latency is higher, the AP overheads are very low which shows that the NP is successful in offloading the communication related tasks from the AP. The time required for 1024 byte Datasend operation is 7.00 msec. compared to the 8.00 msec. reported for a MoveTo operation [12].

7 Conclusions and Future Work

The preliminary version of HARTOS provides facilities for communication between processes in a distributed real-time computer system. We have implemented communication via short messages, signalled events, and data transfers. The implementation has shown the costs involved in providing the communication support. Through our timing measurements, we have been able to identify the most time-consuming operations. Hence, we can work to optimize these operations. We have also seen that, taking into account architectural differences, HARTOS performance is comparable to other message-passing systems like the V system. These architectural differences result in increased communication latency. However, they reduce the communication overhead on the AP and allow for more power and flexibility within the HARTS nodes. The implementation has also provided feedback for the NP design. For example, it is desirable to have hardware support for buffer management and interfacing to the network device (routing controller) to reduce the setup time for packet transmission and reception. Also the NP processing time requirements indicate that a more powerful CPU would significantly reduce the latency of communications.

We now have a framework in place to develop distributed real-time applications. One such application currently under development is the Synthetic Workload Generator (SWG). The SWG will allow the generation of a synthetic workload on a distributed system. It is based on the hierarchical workload model and uniprocessor SWG developed by Woodbury [13].

The SWG is a tool which provides automatic generation of a workload based on structure and behavior parameters provided by the user. For example, one can specify the location of tasks within the system, the frequency of task execution, task behavior, and the task scheduling discipline. The generated workload consists of a set of user-defined application tasks and a set of system control processes. The application tasks may be activated periodically or asynchronously, and may make use of all pSOS and HARTOS calls. The system control processes manage the application tasks. They activate the tasks at the appropriate times, schedule them, and collect data from them. The system control processes are located on each processor and manage only those tasks which execute there. However, they rely on the HARTOS communication facilities for synchronization between processors.

A major application of the SWG is to create workloads with differing degrees of demand on the processors and communication systems. Such workloads will allow us to evaluate the performance and dependability of the system more thoroughly. We will also be able to produce a workload which would approximate a "real" workload in a given application domain, e.g., avionics control. The workload would exhibit the behavior of a real application with respect to parameters such as CPU usage and communication requirements. Data collected during the execution of the workload could be used for evaluating the performance of a given system configuration within the requirements of the application. It would also allow us to calculate such values as the probability that a task will not complete before its deadline (*dynamic failure*)[14]. By using such measures, we will be able to compare various design and implementation approaches in order to optimize the system for real-time applications.

8 Acknowledgement

The authors would like to thank former and current members of RTCL for their comments and inputs to the design and evaluation of HARTOS. They are also grateful to Andre van Tilborg at the ONR for his technical and financial support of this work.

References

[1] M.-S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, routing, and broadcasting in hexagonal mesh multiprocessors," to appear in *IEEE Trans. on Computers.*

[2] J. W. Dolter, P. Ramanathan, and K. G. Shin, "A VLSI architecture for dynamic routing in HARTS," Technical Report CRL-TR-04-88, Computing Research Lab., The University of Michigan, April 1987.

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

[4] T. B. Smith and J. H. Lala, "Development and evaluation of a fault-tolerant multi-processor (FTMP) computer volume 1: FTMP principles of operation," Contractor Report 166071, NASA, May 1985.

[5] A. Olson and K. G. Shin, "Message routing in HARTS with faulty components," submitted for publication.

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

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

[8] D. R. Cheriton, "VMTP: A transport protocol for the next generation of communication systems," In *Proc. SIGCOMM 86*, pp. 406–415. ACM, August 1986.

[9] M. Gerla and L. Kleinrock, "Flow control: A comparative survey," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 553–574, April 1980.

[10] J. W. Dolter, P. Ramanathan, and K. G. Shin, "Performance analysis of message passing in HARTS: A hexagonal mesh multicomputer," submitted for publication.

[11] *ENP K1 Kernel Software User's Guide*, Communication Machinery Corp., May 1986.

[12] D. R. Cheriton and W. Zwaenepoel, "The distributed V kernel and its performance for diskless workstations," In *Proc. 9th Symposium on Operating Systems Principles*, pp. 129–140. ACM, October 1983.

[13] M. H. Woodbury, *Workload Characterization of Real-Time Computing Systems*, PhD thesis, The University of Michigan, Ann Arbor, MI 48109, August 1988.

[14] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A unified method for evaluating real-time computer controllers and its application," *IEEE Transactions on Automatic Control*, vol. 30, no. 4, pp. 357–366, April 1985.