RSD-TR-1-86

COMMUNICATION TASK APPROACH TO THE DISTRIBUTED REALIZATION OF AN INTEGRATED MULTI-ROBOT SYSTEM

Heung K. Lee and Kang G. Shin

Division of Computer Science and Engineering

Department of Electrical Engineering and Computer Science

The University of Michigan

Ann Arbor, Michigan 48109

This work was reported in part by the NSF grant No. ECS-8409938 and the US Airforce Contract Nos. F49620-82-C-0089 and F33615-85-C-5105. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

TABLE OF CONTENTS

ABSTRACT

A distributed computer system is considered for realizing an Integrated Multi-Robot System(IMRS), which is a collection of robots, sensors, computers, and other real-time devices used in contemporary industrial automation. Using an extended port as a data structure, we propose the concept of a ***communication task*** as a ***port manipulator*** to support not only inter-communications but also time handling in the IMRS. This concept remedies the usual limitation of communication models based on message passing.

Implementation of IMRS communication primitives is first discussed on the basis of the extended port. Then, we present a hierarchical structure of the communication task which is suitable for supporting both intra-node and inter-node communications in the system. Various port options have made one-to-many, many-to-one, and many-to-many communication systems possible.

Index Terms - IMRS, real-time distributed systems, communication task, port, communication primitives.

1. INTRODUCTION

The concept of an Integrated Multi-Robot System (IMRS) was first introduced in [1,2] with a specific goal of reducing the usual communication bottleneck and unreliability of a central controller, which is most commonly used in contemporary integrated manufacturing systems. The IMRS is a collection of robots, NC machines, transport mechanisms, sensors, and computers which operate in real-time to accomplish industrial processes. In the IMRS, an industrial process consists of subprocesses, each of which can be programmed with a software *module*. Further, each module is decomposed into several computational *tasks*.[1] Coupled with this natural decomposition, the availability of inexpensive microprocessors and memories with remarkable capacity makes it attractive to realize the IMRS with a distributed computer system. Such a system can provide a high degree of concurrency and reliability through the multiplicity of processors and memories. The distributed system will allow cooperating IMRS tasks to execute in parallel while communicating via message passing.

One of the basic issues in the design of a distributed computer system is intercommunications among modules, tasks, and processors. The nature of such communications is closely related to the communication system and the interconnection network to be used for their implementation. In this report, we consider a communication system and their implementation for an IMRS, provided that the system uses a passive-link interconnection network. Reconfigurable multi-stage networks are an alternative but are excluded from consideration due to their requirement of additional switching circuits and complexity of dynamic control.

[1]The term "process" will be used here to denote industrial output, whereas the term "module" or "task" will be used to represent a computational entity.

To maximize concurrency and provide clean interfaces between tasks in a distributed system, port-directed communications were proposed [3,4]. Concurrent tasks make a reliable rendezvous difficult, since a task should be ready for the concurrent message read/write according to the other task's status. Ports have appeared as specialized memories, namely buffers to separate the receiver of the message from the addressed entity [5,6]. Port is always ready to accept input or deliver output and, thus, a task can communicate with others without blocking (time delay) and risk (message loss). The strict synchronization requirement can thus be relaxed by the use of port. However, since the use of port introduces additional overheads, e.g., scheduling delay and an extra level of indirection, the usefulness of port was questioned in conventional computer systems [7,8].

Cooperating tasks can attain a maximum concurrency by assigning concurrent tasks to different processors. Use of a distributed computing system and the requirement of real-time handling of messages make the IMRS environment different from those in [7,8]. First, some additional functions, such requirements as network communications and time handling of messages, are necessary. We extend port with various options to satisfy these functions. The indirection problem becomes insignificant since messages should be handled via port for these functions. Second, the critical section of port becomes large as the number of its functions increases. Thus, even if the code and data needed for port operation were located in the same processor, the scheduling problem becomes easier.

Although some work has been done to use ports for distributed environments [9,10], little has been done to support a real-time environment. Both of [9,10] focused on the implementation of I/O commands in guarded regions for nondeterministic read/write [4].

The nondeterminism is usually needed for the true parallelism [11]. However, such a nondeterminism is not usually allowed in a real-time environment due mainly to possible communication deadlocks (thereby not meeting time constraints). The real-time environment makes the interface for inter-task communications complicated due to the need of time constraints of messages and that of message scheduling to minimize rejection of messages caused by the promptness control [12]. Inter-task communications in a real-time system are usually deadline-oriented and have time-dependent priorities. We will use a hierarchical network model for port operation; port must have a function of network communication interfaces. Messages will be processed on the basis of priorities which in turn depends on their timing constraints, i.e., communication nondeterminism is disallowed.

In this report, we will consider the implementation of communication primitives in [2] by using extended ports and propose a communication system for an IMRS. The communication system will be based on ports which provide two distinct advantages: programming ease and flexibility, and efficient communications interface between cooperating tasks. The proposed communication system is particularly well-suited for a distributed real-time implementation of the IMRS.

The report is organized as follows. In Section 2 we review briefly our previous work for completeness. Section 3 proposes the structure of an extended port, and Section 4 discusses the port-based implementation of the communication primitives proposed in [2]. In Section 5, we will examine the software architecture of the communication system, which contains *Communication Task*, *Port*, and *Timer*. A hierarchical structure of the communication task is also described in the context of network communications. The report concludes with Section 6.

2. IMRS PROCESS CLASSIFICATION

Before delving into extended ports and the IMRS communication system, it is necessary to review briefly our previous work in [1,2].

As mentioned in the Introduction, the term "process" will be used to mean an *industrial* (but not computational) process, which could be decomposed into several *subprocesses*. Each subprocess may be accomplished by executing a *module* in a computerized controller. Each module can be decomposed into computational *tasks*.

We group IMRS processes into four classes as shown in Table 1. Each process is broken into two or more subprocesses, whose intended work may or may not be dependent. The actions taken (in both the software and hardware) to achieve each subprocess also may or may not be dependent.

Subprocesses	*Actions*	*Process Class*
Independent	Independent	Independent
Independent	Dependent	Loosely-Coupled
Dependent	Dependent	Tightly-Coupled
Dependent	Independent	Serialized-Motion

Table 1. The four basic process classes.

In Table 1 we have named each of the four possible process classes appropriately. The formal definitions of each process class conform to the different interactions between subprocesses and their actions. Examples of each class are:

A. *Independent Processes:* Two robots exist on the same plant floor, but the work for each robot is independent of the other's and is blind to the other's existence. Each robot may depend on common state variables (e.g. conveyor belt). The values of these state variables are determined by many different tasks, and thus simultaneous changes must be handled reliably, e.g., by use of a ***proprietor*** or ***administrator*** in [8].

B. *Loosely-Coupled Processes:* Tool sharing is an example of this class. If robot A is using tool T, another robot B may be forced into either waiting for tool T, or into performing another action not involving tool T. The work of each robot is independent, but the individual actions taken are not. Collision avoidance between two robots executing independent processes but sharing the same workspace is another example of a loosely-coupled process.

C. *Tightly-Coupled Processes:* One example of a tightly-coupled process are two robots which must grab a long steel beam off a conveyor belt. The action of one process must be tightly-coupled to the action of the other process, otherwise the beam could slip or damage could occur to a robot.

D. *Serialized Motion Processes:* We have chosen the name ***serialized motion*** because the most practical process illustrating this interaction involves serializing the action of different robots. If subprocess A must be executed before subprocess B can commence, then A and B form a serialized motion process. The use of one robot as a generalized fixture for another robot is an example of this.

E. *Work-Coupled Processes:* This class is not listed in Table 1 because it is not a basic process class. If two processes are work-coupled, then should one process fail, the other will perform error recovery and take over the responsibilities of the failed process. It is obvious that the process will also be one of the four aforementioned processes. Work coupling may be *one-way* or *two-way*, depending on the ability of the equipment to be used toward either process.

In fact, the above classification reveals naturally communications needs in the IMRS as shown in Table 2.

Process classes	*Communication primitives needed*
Independent process	send/receive and remote procedure call
Loosely-coupled process	query/response
Tightly-coupled process	remote procedure call
Serialized process	signal/wait & multi-way synchronization
Work-coupled process	send/receive

Table 2. Process classes and their communications needs.

Based on the above classification, the *module architecture* -- the structure of a module and communication channels that connect the modules in an IMRS -- was proposed [1]. To support the module architecture, the following communication primitives were also proposed [2].

- **send-receive-reply**: **send** and **receive** are blocking primitives, whereas **reply** has non-blocking semantics. With these primitives, both blocking and non-blocking semantics can be attained.

- **query-response**: **query** is similar to a *remote procedure call* (RPC) [13], except it causes an interrupt to the other task. However, the execution of **query** depends on whether or not the **query** has a higher priority than the current thread of control.
- **order**: This is a directive to a user programmable scheduler. This primitive allows a task to decide whether to suspend its current thread of execution for an incoming **query** or not.
- **waitfor**: This performs n-way rendezvous, which is necessary for the IMRS. (See [2] for a justification of this need.) To implement **waitfor**, a message will have to be sent to every task in the waitfor list of the task executing **waitfor**.

Although RPC is a popular form of network communications in a distributed system, it is efficient only for the fetch-style operation and does not normally support multicast [5]. For this reason, a rather complex but more general set of primitives is needed as given above.

Implementation of the above primitives will be discussed in Section 4 on the basis of an extended port whose language syntax is the subject of the next section.

3. SYNTAX OF PORT

For clean interfaces between communicating IMRS tasks, we propose the concept of *communication task* (CT). A communication task T_{ic} is associated with an IMRS task T_i and deals with all communication related chores for T_i. More on this will be discussed later.

Using several owner and user options, we extend the functions of port to allow programming flexibility and efficient inter-task communications. Although port was originated from multi-clients and single server communication systems, port options can make it useful to more general systems. Fig. 1 shows the language syntax of port which is nothing but a data structure for a communication task.

```
    owner: port port_id param_list {owneroption}
    user : use_port port_id param_list {useroption}

owneroption ::= usage = usages
              | message_format = record_type
              | user_list = (task_or_mod {; task_or_mod})
              | filter = task_id
              | time_out = numeric_const timeunit
              | class = numeric_const
              | bounded_buffer[numeric_const]
useroption ::= usage = usages
              | time_out = integer timeunit
usages ::= send | receive | reply | query | response | waitfor
task_or_mod ::= task_id | mod_id
timeunit ::= msec | sec
```

Figure 1. The language expression of port (BNF form).

As shown above, port describes the characteristics of inter-task communications. Once a port is declared in the user's program, the compiler creates a globally unique port identifier (PID) and a table of task locations for routing messages among tasks. The compiler also checks whether there are duplicated port names each of which is owned by different tasks. In the port expression of Fig. 1, usage, message_format, and user_list are checked for their compatibility at compile time for an efficient run-time implementation, whereas the options shown in Fig. 2 are checked at run-time. (Note that these two types are not disjoint.) CT uses port *P* with these run-time options not only for receiving and interpreting messages, but also for sending messages.

Owner options	*User options*
usage	message_format
user_list	time_out
time_out	usage
message_format	class
class	
filter	
bounded_buffer	

Figure 2. Owner and user options of a port.

Bounded_buffer is used for storing messages for the owner task of P. Once CT (say T_{ic}) receives a message, it decides whether T_i is the receiver or not. If T_i is the receiver, the message will be stored by T_{ic} in bounded_buffer. Otherwise, the message will be stored in another buffer called ***transit_buffer***.

The usage option specifies the sender and the receiver of a message. With this option and the declaration of another port with the corresponding usage, bidirectional communications are accomplished. Suppose T_A owns port P, and T_B and T_C are the users of port P. For example, we can specify **query - response** communications by user_list $= \{T_B, T_C\}$, and usage of owner = **query** and usage of user = **response**. In this case, port P has one sender and two receivers, and, thus, the port owner T_A can transmit a **query** message to either T_B or T_C, or both of them. If there is no receiver specified, the message will be sent to all of the port users.

In an IMRS, inter-task communications are necessary to forward data, synchronize tasks, or request data. The one-to-many (one sender and many receivers) communication in an IMRS is different from that of conventional concurrent systems, e.g., one I/O driver process and many printer processes [14]. The one-to-many communication in the IMRS means that a sender transmits duplicated messages to many receivers. The usage option supports convenient one-to-many, many-to-one, and many-to-many communication interfaces.

The message_format option is used for programming flexibility; message formats between certain tasks can be different from others, and CT must be able to handle several different types of messages. Message_format is syntactically record variants declared by the user:

```
type message is
/*the user can declare typed variables which are used for parameters of
inter-task communications.*/
    record
      number : integer;
      speed : real;
        .
        .
    end
```

If the user does not declare message_format, the default length (one word) is assigned to each parameter in a communication primitive whose syntax is similar to a procedure call.

The time_out value of Fig. 2 indicates priority for the messages in bounded_buffer; the message with a smaller time_out will be given a higher priority. Thus, CT can decide which message is the most urgent. Similarly, time_out is used for transit messages, i.e., the message with the smallest time_out will be routed first. Before sending a message, CT inserts an appropriate time_out value into the message for routing.

Messages can be lost due to contention or network failures; if an acknowledgement from the destination task is not returned until time_out is exceeded, a time_out handling routine will be activated and the source task may re-send the same message. The message with its time_out exceeded will be discarded to avoid duplication of a message. Time_out is also useful for the receiver of a message to determine how long it is allowed to wait for a message to arrive.

The class option is related to message characteristics. If a message loses its meaning after the message was sent regardless whether it is discarded or lost due to a network failure, no reply is needed; this is termed a *class-1 message*. When a message has its meaning only until its time_out is exceeded and the sender waits for a reply, it is called a *class-2 message* and has priority over class-1 messages. This implies that class-1 messages have non-blocking semantics, whereas class-2 messages have blocking semantics. Consequently, if transit_buffer in Fig. 3 is overflowed, class-1 messages will be removed from transit_buffer.

Filter is an I/O routine, and, if necessary, the table concerned with such an I/O communication is stored and used.

Since it expresses the characteristics of inter-task communications, port is created whenever a different communication primitive is required between tasks. However, only one or two ports in each communication task is usually needed because there is high communication locality in an IMRS, and a multi-way synchronization among many tasks is actually realized by **waitfor** as can be seen in the next section.

The concept of extended ports will be applied to the implementation of the IMRS primitives and communication systems in the discussion to follow.

4. IMPLEMENTATION OF PRIMITIVES

In this section, we discuss the IMRS communication primitives in Section 2 on the basis of extended ports.

4.1. Send-receive-reply

These support both blocking and nonblocking semantics as described in [8]. By using a separate communication task, communication protocols can be implemented easily and can thus be made more dependable. Each communication in our system takes only three message transfers. Silberschatz proposed that a port user always send the port owner a signal first to allow communication statements in the guarded regions [4]. This method limits the port's usefulness and requires unnecessary communications. When the port user has **receive** usage, the user should send a signal first to enable the port owner to send a message. It requires four message transfers per communication. In our system, the sending task does not wait until the corresponding receiving action actually takes place. There is no need to perform handshaking before sending a message. The message will be delivered to the destination CT regardless of the destination task's status if there is no network faults or contention. If either the sender or receiver does not receive an appropriate message until time_out is exceeded, a time_out handling routine is used instead.

Consider the case when T_1 and T_2 communicate with each other by **send-receive-reply**. Whenever T_1 wants to transmit a message (to T_2), T_{1c} relays that message to T_{2c}. Once T_{2c} gets the message, it determines whether T_2 already requested that message or not. If it has already requested the message, the message will be sent to T_2 immediately. Otherwise, the message will be saved in bounded_buffer.

We can include timing constraints in these primitives such as how long T_2 could wait for a message from T_1, and how long T_1 is allowed to wait for the **reply** message from T_2. The time_out option in port is used for this purpose. Once T_{1c} gets a message to transfer to T_{2c}, it waits the number of time_out units specified for a **receive** message from T_{2c}. If a time_out exception occurs, T_{1c} activates an appropriate time_out exception procedure in T_1. In case of T_{2c}, it also waits for a **send** message from T_{1c} with the same time constraints. Therefore, when a message arrives, T_{ic} examines the value of time_out in the message for a correct **reply** in time. The remaining time_out value could be used as the execution time of **reply** in T_i.

4.2. Query/response

In general, the RPC paradigm is appropriate for the tasks with master/slave control structures. As a two-message passing statement, **query** is similar to a RPC statement. Thus, **query** is viewed as a **send** followed by a **receive**. On the other hand, **response** is a **receive** followed by a **reply**. However, both **query** and **response** are different from a RPC in that **query** is an asynchronous interrupt of another task [15]. Their control mechanism is more difficult than the conventional message passing, since **query** is not always a more urgent operation than the thread of the task to be interrupted. Thus, whenever the **query** message is sent to a task T_i, T_i decides whether **query** or the current execution thread has higher priority according to the priority specified by the **order** statement. If the current thread has higher priority, the **query** request will be queued. If **query** has higher priority than the current thread, T_i suspends its current thread and execute **query**. After the **query** request has been serviced, T_i resumes the suspended thread of control.

Since **query** can interrupt the current execution thread, when T_{ic} receives a **query** it should be sent to T_i first. If T_i rejects **query**, the **query** message will be saved in bounded_buffer with a time_penalty added. It will then be treated just like the other messages. With an interrupt, **query** is usually used to check the other task's status. When the parent wants to check the status of children, **query** is used with the one-to-many communication scheme. If the destination task is specified, it will be used for one-to-one communication.

4.3. Waitfor

An IMRS needs to have multi-way synchronizations and communications with several other tasks, which are similar to many-to-many multicast communications. In contrast to broadcasting, the multicasting is restricted to some tasks in the waitfor list. Thus, complex algorithms such as the spanning tree forward [16] are not necessary. A separately addressed packet method is used; the copies of the packet are delivered to all destination tasks.

Consider the **waitfor** operation for three tasks T_1, T_2, and T_3. They have their own **waitfor** statements and subsequent functions. When each task gets to the **waitfor**, they can start simultaneously the execution of their functions.

To confirm this operation, once a task gets to its **waitfor** statement, the task sends a message to all of the tasks in its **waitfor** list. If T_1 gets to its **waitfor** statement first, a message containing this fact will be sent to T_2 and T_3, and then T_1 waits for messages from T_2 and T_3. Once messages from T_2 and T_3 are returned, T_1 sends a confirmation message to T_2 and T_3 to make sure that all of the involved tasks have gotten to the **waitfor** statements, and then executes its own function. Even if T_1 received **waitfor** messages from T_2 and T_3, this confirmation action is necessary since

T_1 does not know whether $T_2(T_3)$ received a **waitfor** message from $T_3(T_2)$. It is possible that one or both of them have not received a **waitfor** message due to network faults or other reasons and remain indefinitely in a wait state. The confirmation message prevents this kind of deadlock.

5. THE COMMUNICATION TASK

Ease and efficiency in handling inter-task communications are the key to the success of an IMRS. As discussed earlier, the extended port provides clean interfaces between cooperating IMRS tasks and power of meeting the requirements of real-time constraints and flexible communications. In this section, we address the problem of managing ports with the concept of communication task for a distributed system that realizes the IMRS.

We first discuss the functions of a communication task, and then the CT's structure for network communications.

5.1. Functions of CT

The distributed system for realizing an IMRS consists of a finite number of *nodes*, each of which contains a single or multiple processors. These nodes are connected via a passive network. As mentioned earlier, an industrial process is accomplished by a set of cooperating tasks $T = \{T_1, \ldots, T_n\}$. These tasks are to be executed on a set of nodes $N = \{N_1, \ldots, N_n\}$. Let T_{ic} be the task responsible for inter-communications at the node N_i to which the task T_i is assigned. Hence, T_i and T_{ic} are executed on the same node, N_i. Both T_i and T_{ic} can be located in a single processor or separated by placing T_{ic} on a dedicated processor, called a *communication processor* within the same node [17].

For real-time applications, the system should support various timing constraints, such as time-out, delay, etc. By using another task called the *Timer*, the communication task T_{ic} functions as a ***port manipulator***.

- Port management including intercepting, interpreting, and relaying messages.
- Various systems with one-to-many and many-to-many communications.
- Periodic message updates according to signals from Timer.
- Signaling to T_i if there is a message with the time limit exceeded.
- Error-free transfer of messages to other tasks in the system.
- Failure detection with a timing exception in inter-task communications.
- Network communications including routing and congestion control.
- Message scheduling based on deadline-oriented and time-dependent priorities.

T_{ic} handles chores associated with inter-communications in node N_i. Since it is always ready to accept or transmit messages, it supports inter-node communications via port.

5.1.1. Message Handling

There are two types of message for T_{ic} to handle: those in bounded_buffer, and those in transit_buffer. While the former is for T_i, the latter is for routing messages; different operations need to be applied on them.

As shown in the Appendix, once T_{ic} receives a signal from Timer, it updates time_out values in the messages residing in various ports associated with T_i. If a message with time_out exceeded is found, T_{ic} signals to T_i. Whenever T_i requests a message, T_{ic} should select a message from one of the ports that T_i owns or is authorized to use. If T_i does not specify the message source for T_{ic}, time_out values are used to determine the most urgent message. However, conflict may occur, because there could be more than one port with the same time_out value. To solve this problem, ports are prioritized by the primitive **order**. On the other hand, if the message from the highest priority port is always processed first, then in the worst case an urgent message from a task with lower priority may never be serviced, i.e., "starvation" problem. A scheduling algorithm to solve the starvation problem is thus called for. One solution is to assign a time penalty for each port whose message(s) is rejected because of lower priority. The time penalty will be used to allow all messages to be processed within some specified limit. This time penalty can also be used for other purposes. For example, when a certain **query** message is rejected by T_i since the message has come in via a port with lower priority than the current thread of execution, a time penalty is assigned to that message and saved in bounded_buffer. This prevents continuous trials of a message which cannot be processed immediately, but assures the service of the message within some specified time.

For messages in transit_buffer, time_out is updated whenever Timer signals to T_{ic}. Similarly to bounded_buffer, the time_out value is used to determine which message must be routed next.

5.1.2. Intra-Node Communications

Intra-node communications are quite different from inter-node communications, since while the latter is symmetric, the former is inherently asymmetric, i.e., a client-server relationship between them. Hence, we only consider the speed and efficiency for intra-node communications. There are three tasks in a node N_i: T_i, T_{ic}, and Timer, where Timer operates as a servant of both T_i and T_{ic}. Since the IMRS operates in a real-time environment, it is important to satisfy time constraints. Successful execution of T_i depends not only on its logical correctness but also on whether all related timing constraints are satisfied or not.

Fig. 3 shows a functional block diagram of a node N_i, where the signals between tasks represent their inter-relationship. Since T_{ic} manages messages for T_i, T_{ic} can be viewed as the master of T_i. **Send, receive** and **reply** used for intra-node communications are handshaking signals, <u>not</u> primitives. T_{ic} is always ready to receive signals or messages including those from external nodes.

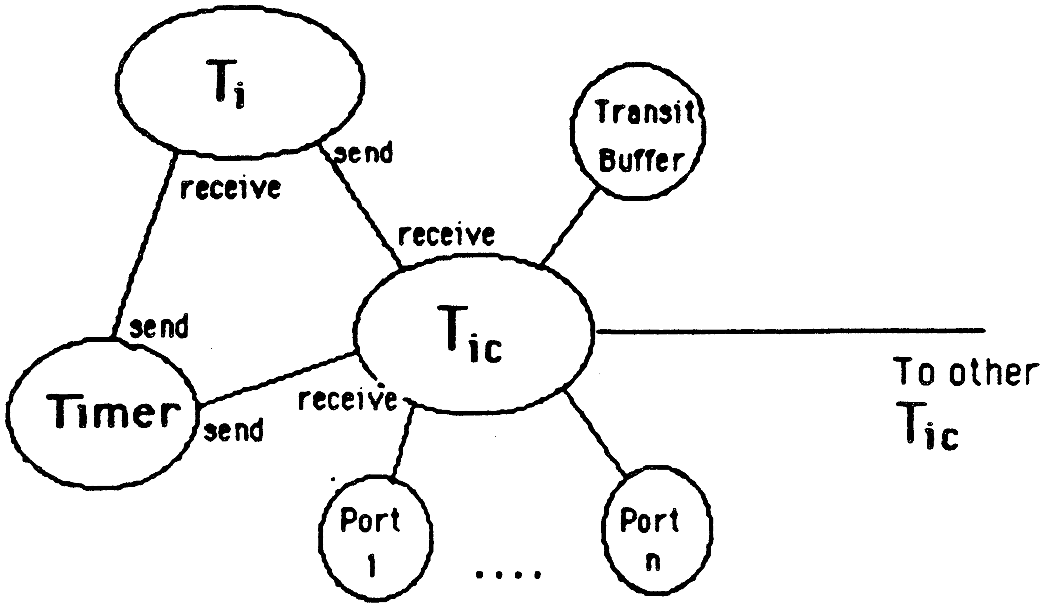

Figure 3. Functional block diagram of a node.

Using the administrator concept in, [8] T_i and Timer always send a **send** signal and are blocked until a **reply** signal is returned. If servants (Timer and T_i) want to write to the master (T_{ic}), they transmit a **send** signal with appropriate data to T_{ic}. Otherwise, they send a "request" signal. This method makes T_{ic} always ready to accept messages.

5.1.3. Time Handling

Usually, T_i has to contain **delay** statements for the industrial process at hand, and T_{ic} needs a clock for passing messages with time constraints. Fig. 4 represents Timer for these purposes, whose function is similar to a hardware programmable timer. It asks both T_i and T_{ic} periodically if they need any timing information; a reply to this must include source identification, time_id and time_length. Time_id is similar to message_id in network communications to identify a request and is assigned whenever both T_i and T_{ic} request timing information. Time_id is also needed whenever several requests are received from either T_i or T_{ic}.

```
PROGRAM TIMER();
 timer = array[ ] of record
                     src_id : integer; /*source_id*/
                     id: integer; /*time_id*/
                     length : integer; /*time_length*/
                   end;
 begin
  repeat
   signal to T_i (send, JOB_REQUEST);
   signal to T_ic (send, JOB_REQUEST);
   if reply is received then
    begin
     -
      the source_id, time_id, and time_length
      are stored in the corresponding timer array
     -
     /*at least there is one timing function request*/
     timer_flag = ONE;
    end; {request from T_i and T_ic }

   /*update the time_length according to the hardware
         unit signal(hardware clock)*/
   if timer_flag = ONE and signal from clock then
    begin

      decrement all time entries in timer
```

```
            -
            if timer[ ].length < 0
              then begin
                send(timer[ ].src_id, TIME_OUT, timer[ ].id);
            -
             removes the corresponding time entries from the timer array
            -
            if timer[ ] is empty
              then timer_flag = ZERO;
          end; {update time entries}

          /*send a CLICK signal to $T_{ic}$ */
          if one system time unit is elapsed
            then send($T_{ic}$ , CLICK);

        until error occur
        - error handling routine -
      end; {timer function}
```

Figure 4. Timer task.

5.2. The Structure of CT

The communication task T_{ic} acts as not only a port manipulator but also as a message relay task for network communications. Thus, T_{ic} is also responsible for routing and scheduling messages. We propose to use a hierarchical structure for network communications, which consists of four levels: *user*, *mapping*, *route*, and *primitive levels*. Each of these levels corresponds to the user program interface, end-node specific in, routing, and adjacent node specific [18]. However, their functions are limited only to support message passing in a distributed computer system, since communications are a large portion of run-time execution. The user level is concerned with intra-node communications, which was discussed earlier. The mapping level deals with address translation, and the route level is concerned with how to route and schedule inter-node messages between tasks. The primitive level is responsible for the physical data transfer between processors. Each of these levels is a procedure which communicates with one another within T_{ic}. When T_i communicates with other tasks, a message will pass through this hierarchical structure transparent to T_i (see Fig. 5).

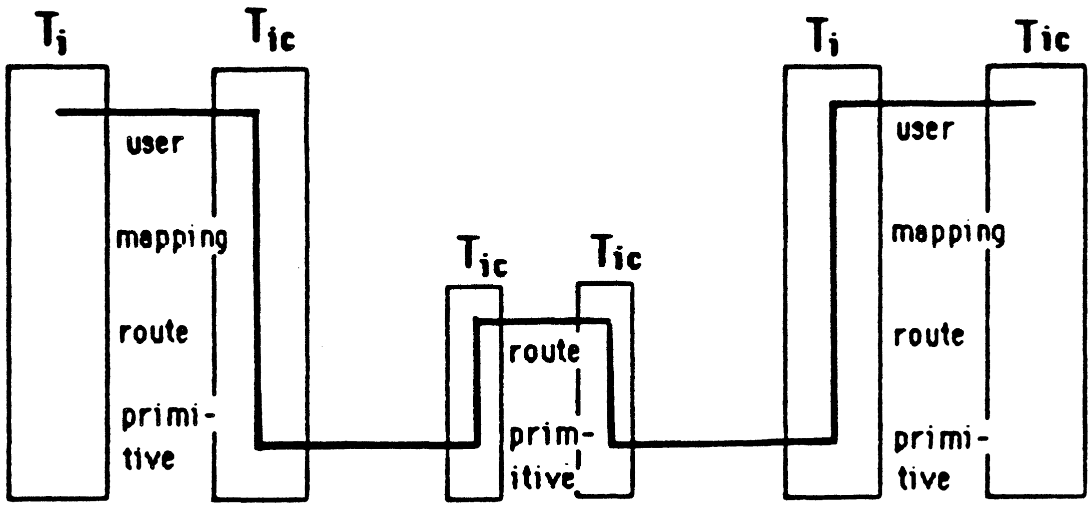

Figure 5. Communication path.

For practical reasons, we assume a packet to be 32 bytes long as in [19] (see Fig. 6). Note that this fixed packet length is convenient for prioritizing messages on the basis of time_out constraints. The syntactic form of a packet is the same as the message_format option of port. The header of a packet contains destination_ID, source_ID, message_ID, etc. Time_out is inserted since it is used for not only routing, but also for the specification of how long the destination task can take to reply to the message. In general, the maximum execution time to be allowed for the destination task is the remaining value of time_out when the message arrived.

header	*message*
destination_ID	contents of message
source_ID	
message_ID	
time_out	
class	

Figure 6. The format of a packet.

5.2.1. User Level

This is the highest level in T_{ic} which interprets messages before they are sent to T_i or to the mapping level. The following forms are the **send** statements in port owner and user tasks:

> **send** port_name({task_id}, param_list);
> **use_send** port_name(param_list);

where task_id is optional. The port owner can have task_id as an option since it can have many users, whereas in case of the port user there is no need to specify the task_id of the port owner (there is only one port owner). If the user is not specified with task_id in the port owner task, all the tasks in user_list are the destination tasks. Otherwise, one-to-one communications are provided.

The **send** and **use_send** statements are translated by the compiler into the following form: **pid**(code, {task_id}, parameter). The code indicates what T_i wants, e.g., send or request a message. The **send** (or **use_send**) statement is omitted since it is interpreted in the destination task by the usage option of port. The T_i's request always has a higher priority than the current thread of T_{ic}; whenever T_i sends a signal, it will be accepted immediately by T_{ic}.

Fig. 7 shows one-to-many and many-to-one communications by using **send** and **use_send**, respectively.

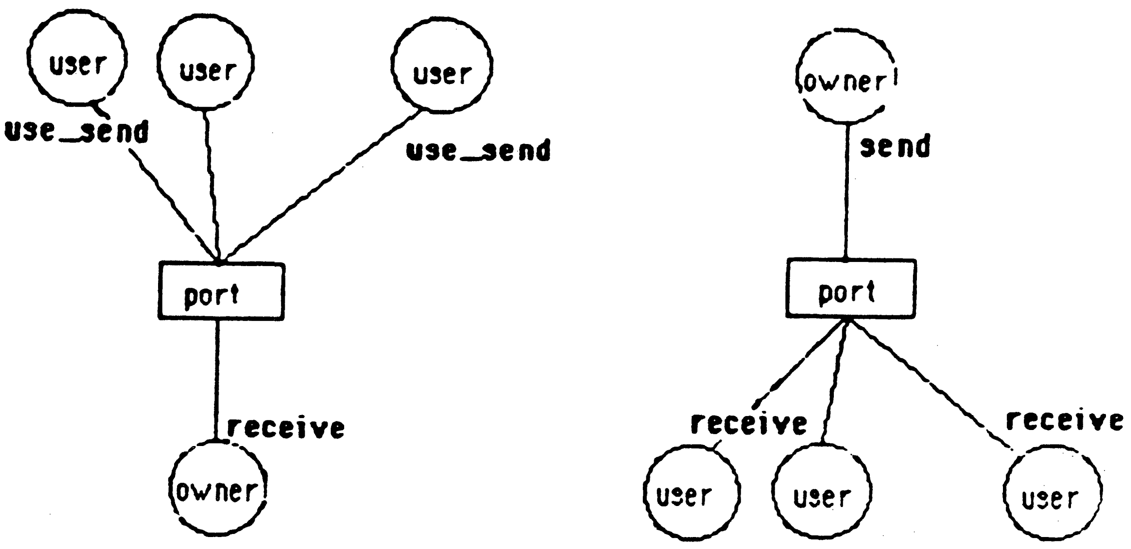

Figure 7. IMRS communication system.

The user level is responsible for an interface with T_i and the mapping level as follows.

```
/*response handler for $T_i$ */
 USER();
  begin
   request ⟶ $T_i$
    begin
     case signal from $T_i$ of
          SEND_MSG : /*sends a message to other tasks*/
           if destination = SPECIFIED
              then begin
```

```
                    -
                      look up ports and make a packet of message
                    -
                    signal to TIMER(reply, request_id, time_out);
                    MAPPING(port_id, {task_id}, parameter);
                end; {specified}
                else begin
                    -
                      find user tasks in the user_lists of
                      ports and make a packet of message
                    -
                    repeat
                      signal to TIMER(reply, request_id, time_out);
                      MAPPING(port_id, {task_id}, parameter);
                    until {the message is sent to all of users}
                end; {unspecified}
            GET_MSG : /*gets a message from port*/
                code = get(port_id, {task_id});
                if code |= NOTHING then
                    send_msg_to_task(port[ ].bounded_buffer[ ].usage);
                    request_msg=ZERO;
                    end;
                    else request_msg=task_id;
        end; {case}
      end; {$T_i$}
  || request → mapping level
      begin
        -
        if message_format of packet is query, or if
        $T_i$ already requested the message, the
        message will be sent to $T_i$. Otherwise, the
        message is saved in bounded_buffer
        -
      end; {input message from other $T_i$}
   end; {user}
```

In the above notation, "signal" is a handshaking for intra-node communications, and the remaining procedure and variables are described in the Appendix.

5.2.2. Mapping Level

This level is responsible for address translation for which a physical address table of logical task names is maintained.

```
MAPPING(port_id, {destination}, parameter)
 begin
   request → user level
     /*physical address translation*/
     dest_id = address(destination);
     ACK = ROUTE(dest_id, source_id, pid,
                 message_id, delivery_delay, class, message);
 || request → network level
     -
     remove tag and call the user level
     -
 end; {mapping}
```

5.2.3. Route Level

A packet called the *datagram* is used for implementation simplicity [20]. Each message is independent of others, and this level is responsible for both routing and congestion control. One can decide with which port each task communicates before running, and communication characteristics do not change dynamically unless a network or task failure occurs. Once a task is assigned, context switching is not allowed until its completion. This means that a processor does not embark on another task before the currently executing task is completed and retired. Under this assumption, we can avoid the problem of run-time allocation of resources. Thus, among various known routing algorithms, static routing, the simplest algorithm, is used at this level. A path from the owner task to a user task can be established *a priori*. Once the path is established, each T_{ic} will maintain information about routing, such as its succeeding node for a specific transit message. However, T_i and T_{ic} may have to be migrated to another node because of network or processor failures. In that case, a new path for those should be found and related tables must be updated accordingly.

For congestion control, we can indirectly estimate the amount of communications between nodes by the time_out specified in port and can allocate tasks according to the characteristics of their inter-communications. Naturally, the tasks which heavily communicate with one another, e.g., vision sensing and processing tasks, should be located in physical proximity. Since each task alone does not have concurrency and has only a single execution thread, T_i does not send several packets of messages to other tasks concurrently. In general, a task accepts (or sends) and processes (or receives) messages sequentially. Their communication characteristics are thus quite different from the general computer network such as ARPANET. Consequently, the congestion and routing problems are somewhat easier to deal with.

```
ROUTE(dest_id, src_id, msg_id, pid, delivery_delay, param);
  node : succeeding node address;
  packet : message containing the above parameters;
  begin
   request → mapping level
    /*message scheduling*/
    .
      update the time_out value in messages and select
      the most urgent message for routing
    .

    /*routing*/
    - find the succeeding node -
    PRIMITIVE(node, packet);

|| request → primitive level
    .
     remove tag and call the mapping level
    .
  end;
```

5.2.4. Primitive Level

This level is the lowest level of T_{ic} and is responsible for message delivery, handshaking and error detection as shown below.

```
PRIMITIVE(node, packet)
  t = 0 : time
  begin
    request → network level
      repeat
        data_link = packet;
        -
         wait until ACK signal is received from the succeeding node
         for fault-free communication or the specified time is elapsed
        -
      until ACK signal is received from the succeeding node
      if error occurs
          then   - exception handling routine -
  || request → other $T_i$
      data_link = ACK; /*send ACK to a neighboring node*/
      -
       remove tag and call the network level
      -
  end;
```

6. CONCLUSION

Using CT, Port, and Timer, a communication system is proposed for an IMRS. The nucleus of the communication system is the communication task which is responsible for network and inter-task communications in a node. For network communications, we proposed to use a hierarchical structure of CT. Timer is used to keep track of time in a real-time environment. The concept of an extended port plays a key role for inter-task communications.

There are several obvious advantages to use ports. One of them is that it can support one-to-one, one-to-many, and many-to-many communications. A task can send a message to all of the port users simply by calling the port name. Another is the message description, by which input messages can be interpreted and output messages can be delivered. A message does not have to contain its own description, making the message handling easier.

Currently a simulator is being developed to investigate various issues including reliability of the protocol, routing algorithms, buffer management, communication delay, and performance. These results will be fed to the development of an experimental testbed at the Real-Time Computing Laboratory, The University of Michigan.

REFERENCES

[1] K.G. Shin, M.E. Epstein, and R.A. Volz, "A Module Architecture for an Integrated Multi-Robot System", *Proc. 18-th Annual Hawaii Int'l Conf. on System Sciences,* Jan. 1985, pp. 120-129.

[2] K.G. Shin, M.E. Epstein "Communication Primitives for a Distributed Multi-Robot System", *Proc. of the IEEE 1985 Int'l Conf. on Robotoics and Automation,* March 25-28, 1985, St. Louis, Missouri, pp. 970-979.

[3] R.B. Kie and A. Silberschatz, "Comments on Communication Sequential Processes," *ACM Trans. on Programming Language and Systems,* Vol. 1, No. 2, October 1979, pp. 218-225.

[4] A. Silberschatz, "Port Directed Communication," *The Computer Journal,* Vol. 24, No. 1, 1981, pp. 78-82.

[5] D.R. Cheriton, "Preliminary Thoughts on Problem-Oriented Shared Memory: A Decentralized Approach to Distributed Systems," *Operating Systems Review,* Vol. 19, No. 4, October 1985, pp. 26-33.

[6] R. Rashid and G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel" *Proc. of the 8th Symposium on Operting Systems Principles,* ACM, December 1981, pp. 64-75.

[7] E.S. Roberts *et. al,* "Task Management in Ada - A Critical Evaluation for Real-Time Multiprocessors," *Software Practice and Experience,* Vol. 11, 1981, pp. 1019-1051.

[8] W.M. Gentleman, "Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept," *Software-Practice and Experience,* Vol. 11, 1981, pp. 435-466.

[9] T.W. Mao and R.T. Yeh, "Communication Port: A Language Concept for Concurrent Programming," *IEEE Trans. on Software Engineering,* Vol. 2, No. 2, March

1980, pp. 194-204.

[10] J.P. Elloy and P. Molinaro, *Port-Oriented Synchronization and Communication in a Local Network.* North-Holland, Real-Time Data Handling and Process Control-II, 1984, pp. 121-129.

[11] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM,* Vol. 21, No. 8, 1978, pp. 666-676.

[12] G.L. LeLann, J.F. Meyer, A. Movaghar, and S. Sedillot, "Real-Time Local Area Networks: Some Design and Modeling Issues," *Technical Reports,* INRIA, Project SCORE, Dept. of Electrical Engineering and Computer Science, U. of Michigan.

[13] A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems,* Vol. 2, No. 1, Feb. 1984 pp. 39-59.

[14] G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys,* Vol. 15, No. 1, March 1983, pp. 3-43.

[15] Y. Parker and J.P.. Verjus, *Distributed Co-operating Processes and Transactions.* Academic Press, Distributed Computer Systems: Synchronization, Control and Communication, 1983, pp. 23-50.

[16] G. Gopal and J.W. Wong, "Delay Analysis of Broadcast Routing in Packet-Switching Networks," *IEEE Trans. on Computers,* Vol. 30, No. 12, December 1981, pp. 915-922.

[17] R.M. Fujimoto and C.H. Sequin, "The Impact of VLSI on Communications in Closely Coupled Multiprocessor Networks", Proc. of Compsac 82, pp. 231-238.

[18] D.R. Kosmalski, "MAP Specification", Technical Note, GM Technical Center, April 1984.

[19] D.R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software,* April 1984, pp. 19-42.

[20] A.S. Tanenbaum, "Network Protocols," *Computing Surveys,* Vol. 13, No. 4, December 1981, pp. 453-488.

APPENDIX

```
/*program $T_{ic}$ consists of one main body and one response handler for $T_i$ */

PROGRAM $T_{ic}$ ();
  port : array[ ]
       of record
            bounded_buffer : array[ ] of record;
                                  msg : array[ ] of parameter;
                                  time_out : integer;
                                  source_id : integer;
                                    ....
                                  end; {bounded_buffer}
            port entries : integer;
                ....
          end; {port}
  node : succeeding node;
  code, task_id : integer;
  request_task : integer;


     procedure get(task_id,code);
       task_id : integer;
       begin
         if task_id is specified
           then begin
                  - search the bounded buffer -
                  end;
           else begin
                  -
                   search a message with the highest priority.
                   if there is no message,
                   then code = NOTHING.
                  -
                  end;
       end; {procedure}


    /*send a message to $T_i$ */
       procedure send_msg_to_task(code);
         code : integer;
            begin
             case code of
                 QUERY :
                   signal to $T_i$ (query,packet);
                   /*wait response*/
                   if response.code = REJECT
```

```
              then save(TIME_PENALTY);
           SEND, WAITFOR :
              signal to $T_i$ (send,packet);
              /*wait for response*/
              if reply.code = REJECT
                 then save(TIME_PENALTY);
        end; {case}
    end; {procedure}

    procedure save(time);
      time : integer;
      begin;
        -
        if time is non-zero, the time_out option of
        message is incremented with that value, and the
        message is saved in a port.
        -
      end;

    procedure save_transit_buffer(packet);
      begin;
      - save a packet in corresponding port -
      end;


begin
  repeat begin
    order($T_i$, TIMER, $T_{ic}$ );
    /*wait for a message from other tasks*/
    source_id = receive(packet);

    case source_id of
      TIMER:
         if timer_code is CLICK
            then begin
            /*update time_out entry of message in port*/
            while update all message do
              begin
                - port[ ].bounded_buffer[ ].time_out decrement -
                if port[ ].bounded_buffer[ ].time_out $< 0$
                 /*signaling to $T_i$ that the message is aged*/
                 then begin
                   -
                   check the message to be aged and remove it.
                   -
                   signal to $T_i$ (port[ ].bounded_buffer[ ]);
                 end;
            end; {while}
         end; {CLICK}
```

```
          if timer_code = TIME_OUT
            then begin
              -
              find corresponding message_id and
              motivates the handling routine.
              -
            end;

        other $T_i$:
        if destination_id = $T_i$
          then PRIMITIVE(); /*my message*/
          else begin /*transit message*/
            save_transit_buffer(packet);
            while transit_buffer |= EMPTY do
              - find a packet with the smallest time_out -
              ROUTE();
            end; {while}
          end; {transit message}
  end; {$T_{ic}$ main body}
until error occur


  ** response handler 1 of message from its own $T_i$, **
  ** time_out in mapping level **

  USER();

  /*it is presented in the text of the report*/
  ......
  return();  /*return to receive status*/

  - exception handler for the fault -

end; {$T_{ic}$ }
```