A FLOATING COMMUNICATION PROCESSOR ARCHITECTURE IN A DISTRIBUTED REAL-TIME SYSTEM

Kang G. Shin and Yogesh Muthuswamy

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

ABSTRACT

The issues involved in providing hardware communication support at each node in a distributed real-time system are studied. First, a general architecture for each node of the system is described. An algorithm for message handling by dedicated hardware called a *communication processor* (CP) is proposed, maximizing the number of requests handled under the various constraints. Finally, a floating CP architecture is proposed to maximize the utilization of the processors at a node and provide greater fault-tolerance in the system.

Index Terms - Distributed real-time systems, communication processor (CP), application processor (AP), communication task (CT), application task (AT), port, window, floating CP and AP.

1. INTRODUCTION

A single computational job can be divided into several smaller tasks, which can then be executed in parallel after mapping them onto different processor nodes of a distributed system. These tasks exchange data and synchronize with each other via message passing between the different processor nodes. Intertask communications can be accomplished by either *handshaking* or using the concept of *port*. In case of handshaking each task waits for the other task to be ready in order to communicate with it. A port is a data structure associated with a task. It can be viewed as a buffer for a task, which serves as an intermediary between a task and its environment, and supports such communication mechanisms as many to one, one to many and one to one. The use of ports makes the communication between tasks more structured and provides a cleaner interface, thus increasing the modularity of the system.

However, there are a number of difficulties associated with the implementation of port-based communications. Ideally, a port should always be ready to send/receive messages and there should be no message loss in the system. Avoiding the message loss is impossible in a real system because of finiteness of buffer sizes and bandwidths of interconnection networks. The time delay is also unavoidable because of the extra level of indirection introduced with the actual implementation of the port, the extra memory accesses and control mechanisms needed to operate on the port. Messages may also be lost due to network contention, and node and link failures. In a distributed real-time system each message usually has a *deadline* associated with it. If a message is not delivered before the deadline, message loss or, even worse, system failure could follow.

The work reported in this paper was supported in part by the NASA under grant NAG-1-296 and the National Science Foundation under grant DMC-8721492. Any opinions, findings, and recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of the funding agencies.

The concept of port to implement intertask communications was first proposed by Silberschatz [1]. This concept was extended in [2] to implement intertask communications in an integrated manufacturing system (IMS), which in essence is a distributed real-time system. The concept of a logical communication architecture for an IMS was proposed initially in [3]. Despite the difficulties mentioned above, we advocated in [2] port-based communications based on their increased level of concurrency and various port restrictions suitable for real-time applications, e.g., deadline specification. In [2], some low level communication primitives and their supporting language syntax were also proposed. The concept of a communication task (CT) as a port manipulator was introduced in [4]. The use of a *message processor* in order to speed up communications in a general distributed system was proposed in [5]. However, this work neither considered real-time applications nor attempted to devise an efficient hardware architecture to support port-based communications.

In this paper we shall propose an efficient means of mapping the port-based communication architecture to a distributed system with hardware communication support at each node. The architecture is suitable for real-time applications because the time overhead incurred by the execution of CT is reduced by overlapping, to the maximum possible extent, the execution of CT on a dedicated processor called a *communication processor* (CP) with that of the application task (AT) on an *application processor* (AP).

The functionality of CP is analyzed. An algorithm is proposed for the handling of communication requests by the CP, which attempts to maximize the number of requests serviced by considering task priorities, estimated service times, and so on. A general architecture for a *floating* CP (AP) is proposed along with a protocol for reconfiguration. This architecture improves the performance of each node by providing more processing power for applications or communications, whichever becomes the performance bottleneck. Thus, the "floating architecture" can adapt itself to either computation-intensive or communication-intensive problems. The fault-tolerance of each node can also be improved by this architecture, because if one processor fails within a node, the other processor in the node can take it over and function in a gracefully degraded manner.

The paper is organized as follows. Section 2 describes the CP, its functionality, and its relation to the other components within a node. An algorithm for message handling by the CP is also described. Section 3 describes the floating CP (AP) architecture. A protocol for reconfiguration is also described. Section 4 concludes the paper and addresses extensions for future work.

2. THE COMMUNICATION PROCESSOR

A general node architecture involving the Communication Processor (CP) is depicted in Fig. 1. (Note that a few more connections than those shown in Fig. 1 are needed to realize the floating architecture.) The node processing power has been divided between the communication and the application processors. Whenever the AP comes across a task requesting communication, that task is put in a *service queue* at the CP along with the timing details[1] for service.

Let there be M independent tasks executing at a node. These may be executing on a single or multiple physical processors. For the simplicity of analysis, the number of tasks is assumed to remain constant within the period of interest. The tasks begin execution with some kind of pre-assigned priorities which can be changed later by the CP. The M tasks request the CP for service via M independent queues. The CP selects an appropriate request to service from the M requests. (An algorithm to select the appropriate request will be discussed later in this section.) Upon completion of service by the CP, the task is returned via a *ready queue* to the AP. The task can then be scheduled back onto the AP. When a task is blocked waiting for its service to complete, other tasks can be scheduled on the AP, thereby increasing the concurrency in the system. The mechanism for rescheduling the task back onto the AP, once its request has been serviced, will not be discussed here.

The *physical* ports $P_1, \ldots, P_N$ act as receiving *windows* for the messages from other nodes. Note that these are called windows to avoid any confusion with logical ports, each of which is a data structure of a task. The various ports declared by the task resident in a node can then be mapped onto the windows. The mapping of the ports to the windows can either be done statically or dynamically. Each port associated with a particular task may always use a particular window (static case), or it may use any window available at that particular time (dynamic case). The windows can also be treated as a resource which is acquired by the task currently executing on the AP, with a certain number of windows reserved for the incoming message traffic at the node. The exact nature of the scheme used would depend on the traffic patterns and the communication characteristics of the various tasks in the system.

There are three logical buffers associated with each node: the windows, the *transit* and *internal* buffers. The optimum number of windows and buffer sizes depend on the message traffic in the system. A smaller (larger) buffer size might lead to message loss (buffer under-utilization). The processing power of the two processors has to be such that they work in a balanced fashion between the CT and ATs.

Messages at a node can either be inbound or outbound. The CP is responsible for handling both types of messages. Handling outbound messages is relatively easy, since the time at which this has to be done is deterministic. In case of inbound messages a degree of asynchrony is involved; the exact time of their arrival is uncertain.

When a packet arrives at the window, and is destined for that node, it is put in the internal buffer. If it is not destined for that node, it is put in the transit buffer on its way to the destination. The *arrival* of a message at its destination node calls for the assembly of all the packets of the message, and the recovery of the message by the CP. The packetization of messages is also done by the CP.

On the arrival of a message, the CP may interrupt the AP indicating the arrival of the message. At this point the addressed

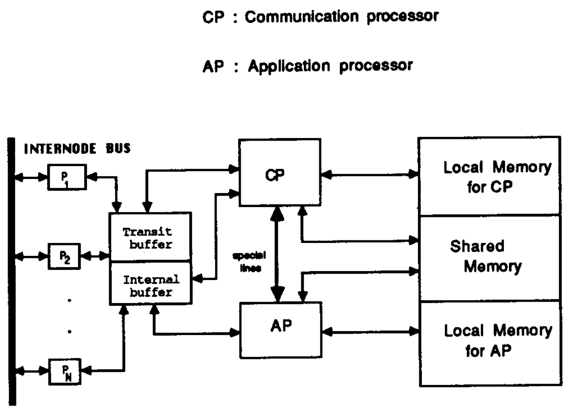


Fig. 1. The organization of a node.

task on the AP may or may not be ready to accept this message. An alternative approach would be to let the tasks executing on the AP queue their requests for message service, i.e., either to send or receive, and let the CP take care of the requests. The CP has to *scan* the list of requests from all of the M tasks at the node before coming to a decision. As soon as the CP comes across a receive request, it can scan the internal buffer to check whether the message has arrived or not. Adopting this approach clearly involves treating send and receive requests asymmetrically. Another option would be to let the tasks provide some kind of *estimated service time* along with their requests and then let the CP select the appropriate request based on some criteria, treating both send and receive requests symmetrically. If there is heavy message traffic, the interruption of the AP by the CP could become excessive. Moreover, the CP then becomes the slave of the AP. This is not desirable, as the CP is envisioned to be more of a node controller. Asymmetric handling of the send and receive requests may introduce an excessive overhead and does not reflect on the priority of the requests served. The receive requests are accorded priority over the send requests. Hence, the scheme of treating the send and receive requests symmetrically is chosen here, and an algorithm to select the appropriate request will be presented later. Note that this scheme also takes care of the asynchrony in the receive requests. If the requested message has not arrived in the specified time span, the concerned task can either queue the request again, or take other actions, such as a time-out recovery.

Handling outbound messages is relatively easy. After an appropriate request to be serviced has been selected, the structured message with its header, text, timing details, and destination is put into the window. The message has to be in a packet form, which is transparent at this stage. These packets have to be reassembled before being put into the internal buffers at their destination nodes. Some routing function has to be built into the window so that the packet may be routed to the next node. This requires intelligence at each window so that the transit messages are routed without the CT's intervention. The window could have a finite state machine, which will compare the address on the header of the packet with the address of all the ports associated with the tasks executing on the node. If the packet was destined for the node, then it places the packet in its internal buffers, else the packet is forwarded to the next node.

[1]These can be specified as port restrictions as described in [2].

The local memory of the CP in Fig. 1 stores the various routines required by the CP to implement its functions, whereas the various application tasks are stored in the local memory of the AP. The AP and CP communicate via shared memory.

In case of a single AT per node, there is a corresponding CT executing on the CP. The CP need not make a special effort to identify which particular AT it is servicing at a particular point in time. The CP's main functions include receipt and disposal of messages, intra-node communications, and time-handling for real-time tasks. Extensions could be made to fault handling. For example, if, due to some transient fault, the message failed to reach its destination, the CP could retransmit the message after a certain period of time.

2.1. An Algorithm for Selecting Requests

Each task resident at a node executing on the AP has its own independent queue to the CP at that node where the task queues its message service requests to the CP. An algorithm is necessary for the CP to select an appropriate request from these queues. Assume that within the queue for a particular task the requests are serviced on a first-come-first-serve (FCFS) basis. The requests are assumed to be queued up in chronological order. Thus, the CP has to select a request from the head of one of M queues. The CP cannot usually draw up a detailed schedule in advance because the list of requests changes dynamically and the selection is priority-based. The most important request at each instant is selected over the less important requests which may have arrived earlier in time.

Each request is assumed to have a service time estimate, deadline, and a priority (might be the same as the one associated with the issuing task). Some simple schemes are to select the request with the highest priority, or the one with least time to elapse of deadline, or the one with minimum service time. In the algorithm described below, it is assumed that the priorities of the requests reflect the time criticality of the various tasks. The $m \le M$ requests,[2] one from the head of each queue, are arranged in order of decreasing priorities. If they have the same priority, the request with the smaller service time is placed first. The first feasible request is one, which, if selected and serviced, will *guarantee* the completion of all higher priority requests than the selected request in time. If the service times estimates are exact, then this scheme is optimal in terms of the number of requests serviced, subject to the constraint that requests with a higher priority than any of the requests serviced shall not remain unserviced.

For the convenience of presentation of the algorithm, the following notation is introduced.

M :	the number of tasks resident at the node.
t :	the time instant a request to be serviced is selected.
$d_i(t)$:	the deadline of request i.
$s_i(t)$:	the estimated service time of request i.
$p_i(t)$:	the priority of request i.
$C_i(t)$:	the estimated completion time of request i, i.e., $C_i(t) = s_i(t) + \sum_{j \le i} s_j(t)$,
FS_i :	The ordering of requests with priorities higher than that of request i such that for each task $k \le i$ in the ordering, $C_k(t) \le d_k(t)$. In other words, this is a feasible schedule for request i.

[2]Because some of M queues may be empty, the number of requests to be scheduled at a given time for service by the CP is less than or equal to M. But, we will always assume M requests to be scheduled, making error on the safe side.

Also, let $S_t^{\,i} = \sum_{j=i}^{M} s_j(t)$ and $D_t^{\,i} = \max_{j \ge i} d_j(t)$.

The lowest priority request is chosen and an attempt is made to build a feasible schedule (in any order of request priorities) from the other requests which have priorities higher than the chosen request using the closest deadline scheduling policy. If this attempt fails, then there is no way to satisfy this request without sacrificing some higher priority request. Hence, the request is dropped from further consideration, and suitable indication given to the task which generated this request. If this attempt succeeds, then that request is chosen for servicing at that particular time instant.

Proposition 1: For any FS_i, $t + s_j(t) \le d_j(t)$ for all $j \le i$.
When we choose a request i, the estimated service times of all the requests whose priorities are higher than that of i should be such that if any of them is scheduled at the current instant, they should complete before their deadline. This is because if they did not do so, we would have an ordering which does not satisfy the requisite condition for a feasible schedule.

Proposition 2: For any FS_i, $D_t^{\,i} - t \ge \sum_{j \le i} s_j(t)$.
If this proposition does not hold, there would be no way to satisfy all requests with higher priority than i, and hence the schedule is no longer feasible.

Proposition 3: Suppose there is a set of requests whose deadlines can be met. Then, the schedule in which the request with the earliest deadline is scheduled first always meets all deadlines.

The above result is borrowed from the work by Jensen *et. al*6 . Using the above three propositions, the CP uses the following algorithm to select a request from the M queues.

```
1.  Arrange all the requests into a list $PL$ in order of decreasing priorities and index them accordingly.
2.  Select the next request $i$ from $PL$ and set $VT := t$, the instant of selection.
3.  If $D_t^{\,i} - t \le \sum_{j \le i} s_j(t)$, go to step 2.
4.  For any request $j \le i$, if $t + s_j(t) \ge d_j(t)$, then go to step 2.
5.  Arrange all requests into a list $DL$ in order of increasing deadlines.
    5a. Choose the next request $k$ from $DL$.
    5b. If $VT + s_k(t) \le d_k(t)$, then $VT := VT + s_k(t)$, go to step 5a.
    5c. If $DL \ne \varnothing$ then go to step 2 else a feasible schedule has been generated and request $i$ is declared as the chosen request.
```

The algorithm described above selects requests to be serviced on the basis of their priorities, while maximizing the number of requests serviced. The performance of this algorithm will depend very much on the basis for the assignment of priorities. For example, if the priorities were just assigned based on the closest deadline, this algorithm reduces to the closest deadline scheduling. (No attempt has been made here to either define the basis for the priorities, or evaluate its impact on the performance, though, since it is not the main intent of this paper.)

2.2. Functions of the CP

Let AT_α and CT_α be respectively an AT and the CT assigned to node α. In the absence of CT, when another task sends a message to AT_α, it has to wait till AT_α receives the message. One of the main purposes of using a CT is to avoid the overhead caused by handshaking. The sender task now sends its message irrespective of whether or not the receiver task is ready. CT_α receives and then directs the message to an appropriate port of AT_α while servicing a receive request from that task. CT_α has to keep track of how many ports AT_α has and what are the various restrictions associated with them. Each port can be associated with an internal buffer to store the messages which are addressed to the task residing at that node or all the buffers of all the ports could be combined into one in order to minimize space. However, this could lead to the dangerous case of a particular port not having space for its messages. A message arriving at a node may be destined for that node (terminal message) or for another node (transit message).

The terminal messages are handled differently. If AT_α has executed a receive command and is waiting for a message from its sender task, then the terminal message which has just arrived is given to AT_α if the AT_α's receive request is currently being serviced by the CP. Another possible situation is that AT_α has not yet executed the receive command, but a message has arrived at that port. This message is stored in the internal buffer. When AT_α requests a message and this request is selected to be serviced, then CT_α scans the internal buffer and gives the message from the port specified by the task. Since each port does not have a specific slot in the physical memory, the buffer search will have to be exhaustive. In case no port is specified, then some built-in rules, like prioritizing the ports, can be followed.

The CT has to maintain the list of all ATs resident in that node and the ports associated with them along with their modes of usage. This can either be stored as a table or as a tree which might allow dynamic creation of ports as and when the situation demands.

The CP is interrupted every time a message arrives, the arrival of a message referring to the assembly of all the packets of the message. The packet level handling is done by the windows, each of which is a finite state machine, as mentioned earlier. It compares the addresses of all the ports associated with the node, and takes suitable measures, like putting the packet in the internal buffer if it is addressed to a task located in that node, and transferring the packet to the window of the next node if it is not a terminal message. Overflow transit message packets, if any, are handled by the CP.

The *timer* task is the task executing at a node, which is responsible for all the timing information provided at that node. For all practical purposes it can be considered a part of the CT and has to execute on the CP. It should inquire the CT whether it requires timing information and take the required action like signalling to the CT when the time has run out.

The CT is subdivided into the user level, mapping level, routing level, and the primitve level (see [4] for more on this). The user level calls the mapping level to ascertain the destination address. The mapping level then uses the routing function to determine the next node for the message and calls the primitive to send the message. In the present discussion, the user and the mapping level functions are performed by the CP. The routing and the primitive level functions are performed by the windows. Some of routing level functions are also performed by the CP, forming an overlapping zone between the CP and the windows.

The basic mode of communication between the AP and the CP is via shared memory. In addition, there should be provisions for hardware communication between the two processors like a set of flags to signal any abnormal condition[3] which might arise. The fault handling in this system is limited to recognizing the time-out exceptions and calling the appropriate recovery routines. For example, if task A sends a message to task B but does not get the acknowledgement (reply) in the specified time, then the recovery procedure is initiated by the CP which might involve retransmitting the message to B and waiting for a reply if the message is sufficiently important to do that, or ignoring it if the message has lost its meaning after that particular time span.

3. FLOATING COMMUNICATION PROCESSOR ARCHITECTURE

In the node architecture where an AP is assisted by a CP to perform the various tasks assigned to that particular node, the AP may run out of tasks to execute and remain idle. It is also possible for the CP to run out of requests to service and remain idle (this is more likely if the tasks are computation intensive). Note that the CP has been used to improve the performance and the utilization of the AP, removing as much of the communication overhead as possible from it. The situation where the AP is blocked (or remains idle) is undesirable. Moreover, the CP, instead of remaining idle (computation intensive tasks), can relieve the AP of some of its load. This motivates the need for a *floating* CP. In this architecture, there are two equally powerful processors at the node and both of them are capable of executing the application as well as the communication tasks. The central idea is that the system can be reconfigured according to the load at a particular time. For example, if the processor assigned to function as an AP has run out of tasks, then it can *switch* to become a CP and then relieve the other processor of some of the load at that instant. This would remove the probability of a processor having to remain idle while the other processor is overloaded, and hence improve utilization. This approach also improves fault-tolerance of the system, since if one of the two processors fails, then the remaining fault-free processor can function as the AP and CP in a gracefully degraded mode.

The flexibility introduced in terms of a floating CP is not entirely without overhead. Normally, the CP is a specialized processor and need not be as complex and as powerful as the AP. Other problems raised by this approach are the reconfiguration overhead, and the memory contention problems when both the processors are executing the same class of tasks. These issues, though important, will not be addressed here.

3.1. A Reconfiguration Protocol

The architecture of a node is depicted in Fig. 2. A typical node consists of two processors, say P_1 and P_2. Let the state of the system be represented by the ordered pair (a,b) where $a = A$ if P_1 is the AP, and $a = C$ if P_1 is the CP. Similarly, b represents the state of P_2. The system can thus be powered up from any of the possible states shown in the Markov model of the system in Fig. 3. Note that the transitions from (C, C) to (A, A) are not allowed, as at any instant only one processor is allowed to change state. (More on this will be discussed later.)

The crucial aspect is the common memory between the two processors (Fig. 2). The memory holds the task control blocks and queues. In this architecture, P_1 (P_2) can take its input from the computation queue if it is functioning as the AP at that instant, and from the communication queue if it is a CP at that time. There are

[3]For example, failure may occur due to time-out exception.

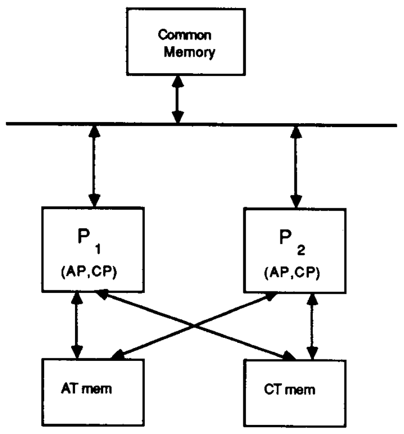


Fig. 2. Floating CP (AP) architecture.

two queues, *CT queue* for communication requests and *AT queue* for computation requests, both kept in the common memory. The queue locations are known to both the processors. If a processor is an AP at any instant, it takes its input from the AT queue and when the task generates a communication request, it enqueues the request onto the CT queue.

If a processor is functioning as an AP at a particular point in time, it selects the first encountered AT requiring computation service from the AT queue. In case the AT queue is empty, it switches to become a CP, and looks in the CT queue for the next task (requiring communication service) to execute. Both the queues cannot be empty, as it is assumed that the number of tasks at a node is non-zero and constant.

The reconfiguration might be as simple as reloading the contents of a few registers, since the processors are identical. When the processor acting as AP discovers that there are no ATs to be serviced, it can issue a supervisory call to the operating system, which will reload the reconfiguration register. Then, the processor will start looking for tasks in the other queue. Thus, the processor looks for a task to execute in a particular queue, depending on the contents of the reconfiguration register. The other additional information regarding where to look for the task code can be incorporated into the task control block.

Since both the AT and CT queues are placed in the common memory of the two processors, only one processor can access a queue at a time, and hence only one processor can switch mode at a given time, i.e., the transitions from (C, C) to (A, A) are not necessary in the Markov model.

Once a processor accesses the common memory, it should take complete control of the memory until it obtains a task to service. This is easy to visualize because, if the other processor is allowed to put a task into the queue when this processor is still looking at the queue, incorrect reconfiguration might take place.

The state diagram of the system is shown in Fig. 3 with the various conditions on which the transitions take place. This can be further simplified by merging states (A,C) and (C,A), thereby resulting in a modified state diagram in Fig. 4. The protocol for reconfiguration is described by the flowchart in Fig. 5. The system is normally assumed to be in state (C,A) or (A,C), which is the state H in the modified state diagram. On encountering exceptions (empty AT queue or empty CT queue), appropriate action is taken. Suppose a processor is the AP at a given time. If the AT queue is

A : AT queue empty B : CT queue empty

C : CT queue non-empty D : AT queue non-empty

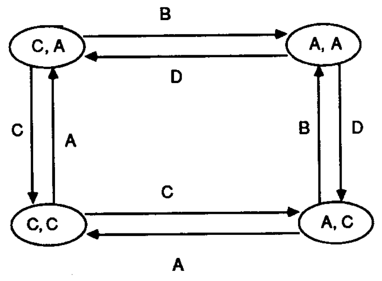
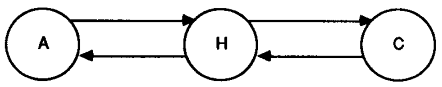


Fig. 3. State diagram for floating CP (AP) architecture.



A : State (A,A) - both AP's

H : State (A,C) or (C,A) - normal mode, one AP, one CP.

C : State (C,C) - both CP's.

Fig. 4. Modified (reduced) state diagram.

empty, it reconfigures itself to a CP and services the next available task in the CT queue. After servicing this task it puts the task back in the AT queue, and reconfigures itself back into the AP. The overhead incurred in this scheme increases with the number of reconfigurations performed during the course of execution.

3.2. Performance Improvement

The improvement obtainable by using the floating CP architecture over the fixed CP architecture is analyzed in this section. The architecture with the fixed CP is always in state H of Fig. 4. In the case of fixed architecture, the states A and C represent situations where one of the processors (AP or CP) is blocked. In these states both the processors are doing useful work in the case of floating CP architecture. Thus, the overall time spent by the system in states A and C represents the improvement obtainable by using the floating CP architecture.

For the purpose of analysis, the model of the system is assumed to be the one shown in Fig. 6. The time spent by the system in states A and C in Fig. 4 is the same as the time spent by the system in states $(0,n)$ and $(n,0)$. The probability of the system being in state $(0,n)$ or $(n,0)$ is given by

$$P(A) + P(C) = \frac{(1-\rho)(1+\rho^n)}{(1-\rho^{n+1})}, \quad \text{where} \quad \rho = \frac{\mu_{ap}}{\mu_{cp}}.$$

The fractional improvement of the floating architecture over the fixed architecture is plotted in Fig. 7. This plot reveals the sharp improvement in performance possible when the number of tasks at the node is small or the server ratio is small. This can be explained

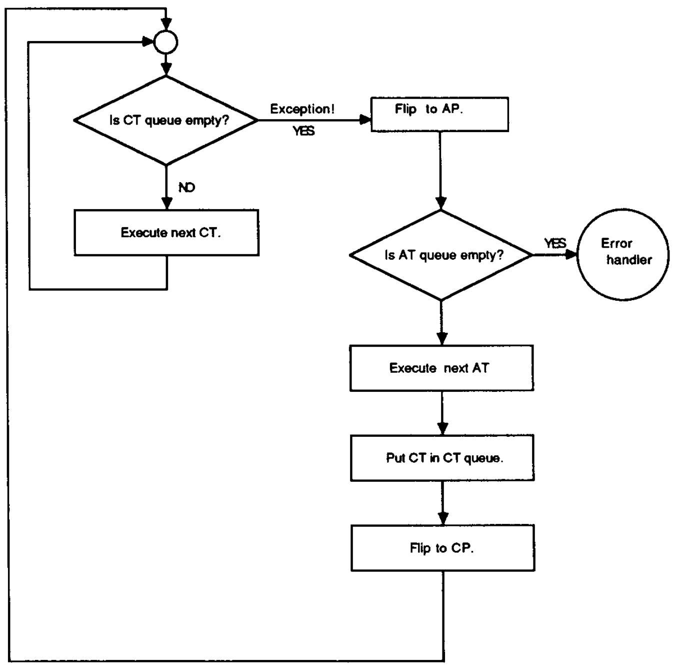
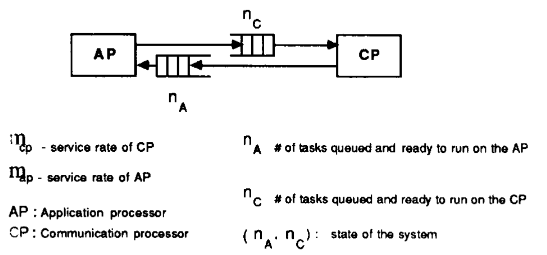


Fig. 5. A reconfiguration protocol for CP.


AP
CP
n_C
n_A
m_{cp} - service rate of CP
m_{ap} - service rate of AP
AP : Application processor
CP : Communication processor
n_A # of tasks queued and ready to run on the AP
n_C # of tasks queued and ready to run on the CP
(n_A, n_C) : state of the system

Queueing model of the node.

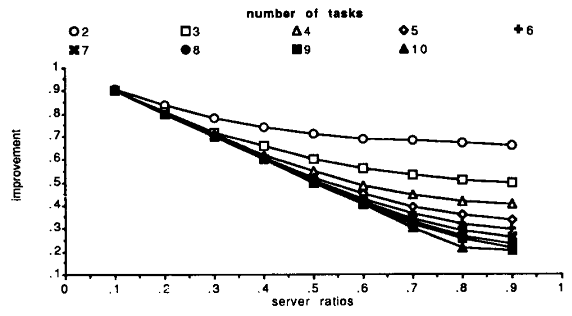
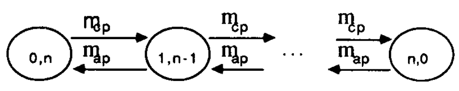


Fig. 7. Performance of Floating CP architecture



Markov Chain model of the system

Fig. 6. Model of the node.

by observing that if there are fewer tasks at a node, they are serviced faster, leading to idling of the quicker processor, i.e., the CP in the case of computation intensive tasks. On the other hand, if the server ratio ρ is small, then most of the time is spent in the AP, whereas the CP remains idle most of the time, and hence the sharp improvement possible by using this approach.

4. CONCLUSION

In this paper, various issues involved in the communications aspects of distributed real-time system were studied. Particularly, addition of extra hardware at each node, in the form of a communication processor (CP), was analyzed. An algorithm for message handling by the CP was proposed, which maximized the number of requests serviced in time after taking into account the priorities of the requests. A floating CP (AP) architecture was proposed, which is different from the communication architectures proposed by others to date. In addition to improving the utilization, the floating architecture can improve fault-tolerance of the system.

Although the work described in this paper provides a good foundation, many issues remain to be resolved before an effective communications architecture can be realized for a distributed real-time system. The floating CP architecture promises to improve utilization and fault-tolerance. However, the overhead introduced by the reconfiguration process has to be studied further. The tradeoff between the performance improvement and the overhead of reconfiguration needs to be analyzed. All of these are matter of our future inquiry.

REFERENCES

[1] A. Silberschatz, ''Port-directed communication,'' *The Computer Journal*, vol. 24, pp. 78-82, 1981.

[2] K. G. Shin and M. E. Epstein, ''Intertask communications in an integrated multirobot system,'' *IEEE J. Robotics and Automation*, vol. RA-3, no. 2 , pp. 90-100, April 1987.

[3] K. G. Shin, M. E. Epstein, and R. A. Volz, ''A module architecture for an integrated multirobot system,'' *Proc. of 18th Hawaii Int'l Conference on System Sciences*, pp. 120-129, Jan. 1985.

[4] K. G. Shin and H. K. Lee , ''Port manipulator for the distributed realization of an integrated manufacturing system,'' *Computer Systems Science and Engineering*, vol. 3, no. 1, pp. 21-31, Jan. 1988.

[5] U. Ramachandran, ''Hardware Support for Interprocess Communication,'' *Computer Sciences Technical Report #667, Department of Computer Science, University of Wisconsin, Madison, WI*, September 1986.

[6] E. D. Jensen , C. D. Locke , and H. Tokuda, ''A time driven scheduling model for real-time operating systems,'' *Proc. IEEE Real-Time Systems Symposium*, pp. 112-122, 1985.