ON THE IMPLEMENTATION AND USE OF ADA[1] ON FAULT-TOLERANT DISTRIBUTED SYSTEMS

By

John C. Knight and John I. A. Urquhart

Computer Science Report #TR-85-06A

June, 1985

ON THE IMPLEMENTATION AND USE OF Ada[1] ON FAULT-TOLERANT DISTRIBUTED SYSTEMS

John C. Knight and John I. A. Urquhart

Affiliation Of Authors

Department of Applied Mathematics and Computer Science

University of Virginia

Charlottesville

Virginia, 22901

Footnote

1. Ada is a trademark of the U.S Department of Defense.

Financial Acknowledgement

This work was supported by the National Aeronautics and Space Administration under grant number NAG–1–260.

Index Terms

Distributed systems, fault tolerance, Ada, highly–reliable systems, tolerance of processor failure.

Address For Correspondence

John C. Knight

Department of Applied Mathematics and Computer Science

University of Virginia

Charlottesville

Virginia, 22901

Abstract

In this paper we discuss the use of Ada on distributed systems in which failure of processors has to be tolerated. We assume that communication between tasks on separate processors will take place using the facilities of the Ada language, primarily the rendezvous. It would be possible to build a separate set of facilities for communication between processors, and to treat the software on each machine as a separate program. This is unnecessary since the existing facilities of the language should be able to satisfy all requirements. The Ada language reference manual specifically states that a system consisting of communicating processors with private memories is suitable for executing an Ada program.

We show that there are numerous aspects of the language which make its use on a distributed system very difficult. The issues are not raised directly from efforts to implement the language. They arise because no semantics are defined in Ada for the situation in which a processor is lost.

Appropriate semantic enhancements to Ada are suggested and extensive modifications to the execution support required for Ada are proposed to implement them. A program structure making use of these semantics is defined that includes all the necessary facilities for programs written in Ada to withstand arbitrary processor failure. Provided the required program structure is used, continued processing can be provided.

I INTRODUCTION

Digital computers are being used increasingly in dedicated control applications that require high reliability. These systems are usually embedded and frequently distributed. Several processors may be used that communicate using a high-speed bus even though they are geographically close. An example is a digital avionics system for a military aircraft in which separate computers may be used for flight control, navigation, displays, weapons management, and so on. The overall system requires some coordination and so the various computers communicate via a data bus. A typical system is described by McTigue [1].

One of the advantages of distributed processing is that a hardware failure need not remove all the computing facilities. If one processor fails, it is possible (at least in principle) for the others to continue to provide service. This is a desirable characteristic for applications requiring high reliability. The use of distributed processing is further encouraged by the decreasing cost of computer hardware.

Ada [2] was designed for the programming of embedded systems (such as those mentioned above) and has many characteristics designed to promote the development of reliable software. In this paper we examine the problem of programming distributed systems in Ada. In particular, we are concerned with the issues that arise when some form of acceptable processing must be provided using the hardware facilities remaining after a failure.

We begin by looking at the general problem of tolerating hardware failure. In section II we present some motivation for considering distributed systems where hardware failure must be tolerated, and define in detail the failures we will consider. In section III we look at the general problem of providing service after processor failure; the facilities needed are described in section IV. In the remainder of the paper we turn our attention from the general problem to the problem of building fault-tolerant systems using Ada. The considerable difficulties that arise when such systems are programmed in Ada are discussed in section V. We show in sections VI and VII that these difficulties can be overcome by careful programming and by making extensive additions to a normal execution-time support system for Ada. These additions make no changes to the language

syntax and their use in Ada is discussed in section VIII. An example of an application program using such mechanisms is given in appendix 1, and a detailed examination of task communication difficulties is presented in appendix 2.

II HARDWARE TOPOLOGY AND FAILURES

The kind of architecture we expect to be in common use for embedded systems in the future in shown in figure 1. It is based on the use of a high-performance data bus that links several processors. Each processor is equipped with its own memory. Devices such as displays, sensors, and actuators would be connected to the bus via dedicated microprocessors. Thus these devices would be accessible from each processor.

Much research has been undertaken in recent years to produce computer architectures of great reliability such as the SIFT [3] and FTMP [4] machines. However, even though designed for reliability, these machines may still fail. Lightning, fire or other physical damage could cause a processor to fail no matter how carefully the processor was built. There is, therefore, good reason for employing software structures able to cope with partial hardware failure.

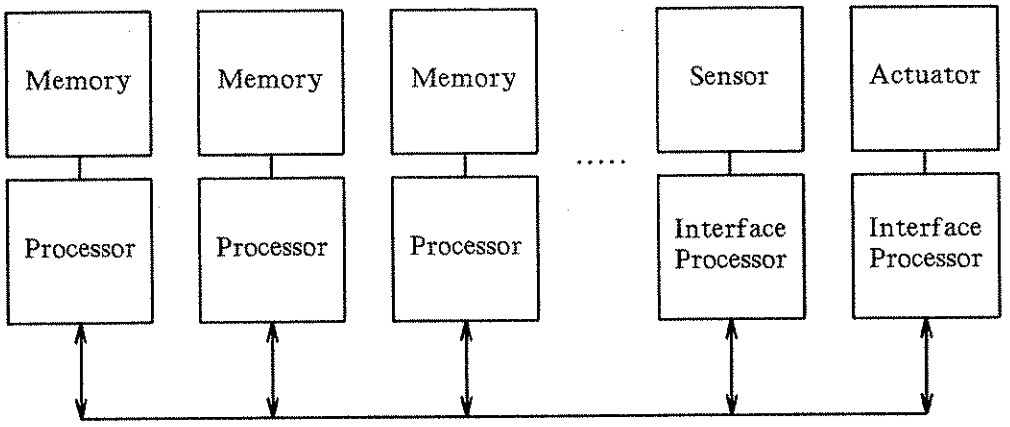

Fig. 1. Distributed Architecture.

It is clear that continued service after failure implies a system distributed over two or more processors. Distribution, however, does not necessarily imply continued service after failure. Several processors sharing a computation that stops whenever any single processor fails will be viewed as a single processing unit. This distinction between distributed systems that allow continued service after failure and those that do not is important. We will call a distributed system that does not allow continued service after a processor fails a *centralized* distributed system. From the perspective of fault tolerance, such a system is no better than a uniprocessor.

We assume that communication between processors on a distributed system will be implemented using a protocol that conforms to the ISO standard seven–layer Reference Model [5]. Such protocols are concerned with communications failure such as dropped bits caused by noise and loss of messages or parts of messages, and we assume therefore that all aspects of message transmission and management are taken care of. The kinds of hardware failure that we are concerned with, the total loss of a processor or a data bus with no warning, are usually not addressed by these protocols and must be dealt with separately.

A processor will be assumed to fail by stopping and remaining stopped. All data in the local memory of the processor will be assumed lost. Thus the case of a processor failing by continuing to process instructions in an incorrect manner and providing possibly incorrect data to other processors will not be considered. We assume that such events are taken care of by hardware checking within the processor or by the methods of Schlichting and Schneider [6].

While this may seem a severe restriction, at least three arguments can be made in its favor:

(1) Faults of the assumed kind must be taken into consideration anyway since a processor might fail in this way.

(2) Either by hardware checking within a single processor or by checking between a dual pair of processors, it is possible for an underlying system to simulate the assumed processor failure mode.

(3) If such a failure mode is not assumed, error recovery becomes extremely difficult. It becomes possible for a processor to fail, and for resulting errors to remain undetected until all data is compromised.

Given this assumption, error detection reduces to detecting that a processor has stopped. Error recovery is simplified by the knowledge that although the data in the failed processor's memory is lost, data on the remaining processors is correct.

A distributed system that is to be highly reliable will be built with a redundant bus structure. Redundancy usually includes replicating the bus along different routes as well as replication of the bus hardware itself on a particular route. Loss of a complete bus need be of little consequence if it is replicated and can be coped with by the low-level communications software. A complete break in the bus system that isolates some subset of the processors (i.e. the network becomes partitioned) is much more serious though very unlikely given multiple routes and replication. The issues that arise in that case are different from those arising from processor failure. They are beyond the scope of this paper and will not be dealt with here.

III APPROACHES TO FAULT TOLERANCE

At the interface provided to the application program, there are two different approaches that can be taken when attempting to provide tolerance to hardware faults. In the first approach, the loss of a processor is dealt with totally by the software providing the interface. Any services that were lost are assumed by remaining processors, and all data is preserved by ensuring that multiple copies always exist in the memories of the various machines. We describe this approach as *transparent* since, in principle, the programmer is unaware of its existence. This is the approach being pursued by Honeywell [7].

Transparent continuation has several advantages:

(1) The programmer need not be concerned with reconfiguration.

(2) The programmer need not know about the distribution. Thus the distribution can be done by the system.

(3) The same program can be run on different systems with different distributions.

However, as the continuation of service is transparent to the programmer, the programmer cannot specify degraded or *safe* [8] service to be used following processor failure. Since the system cannot specify it either, transparent continuation must always provide identical service. If identical service is impossible because insufficient processing resources remain, the system will stop.

In many crucial systems this is not acceptable. Situations will occur where identical service cannot be provided (due to physical damage, say) and yet degraded service is necessary if some

catastrophe is to be avoided. A nuclear power plant may be unable to provide power but nonetheless must be able to shut down safely.

A second disadvantage of transparent continuation is its inefficiency. Since failures can occur at arbitrary times, the support software must always be ready to reconfigure. Duplicate code must exist on all machines and up–to–date copies of data must always be available on all machines. The overhead involved in this process is considerable. Further, the overhead may not be obvious to the programmer when the program is being written. A simple assignment statement, for example, may take considerable time to execute in order to ensure that the updated value of the variable has been distributed to all the memories. However, even if these difficulties could be overcome and transparent continuation could be offered without massive duplication of computing resources, it would still be unsuitable for many applications because of its inability to offer alternate service.

In the second approach to dealing with the loss of a processor, only minimal facilities are contained in the interface provided to the application program. The fact that equipment has been lost is made known to the program and it is expected to deal with the situation. We will refer to this approach as programmer–controlled or *non–transparent* since the programmer is responsible for providing the fault tolerance.

Programmer–controlled continuation has several disadvantages:

(1) The programmer must be concerned with reconfiguration.
(2) The programmer must either specify the distribution or be prepared to deal with any distribution provided by the system.
(3) The program depends on the system; at least the reconfiguration parts do.

The disadvantages are out–weighed by the fact that the service provided following failure need not be identical to the service provided before failure. The programmer can have complete control over the services provided by the software and the actions taken by the software following failure. Alternate, degraded, or merely "safe" service can be offered as circumstances dictate. Also, the various inefficiencies associated with the handling of failure and the necessary preparations for it are quite clear to the programmer.

In the remainder of this paper we will consider only the non-transparent approach. We assume that the actions to be taken by each processor following a failure are specified within the software executing on that processor.

IV REQUIRED CONTINUATION FACILITIES

If a distributed system is to provide continued service after one or more processor failures, then facilities must be provided over and above those needed for normal service. We will refer to these as *continuation facilities*. If there is a single continuation facility for the entire system then the system is centralized. If the processor providing the continuation facility fails, the system will stop. To prevent this, continuation facilities must exist on all the processors.

However, difficulties can still arise if, following the loss of a processor, a continuation facility on one processor is chosen to perform fault tolerance for the entire system. For example, since the processor performing the fault tolerance may fail at any point, all other continuation facilities must be kept advised of the current state of the recovery so that they can take over if necessary. This is unacceptable because of the resulting overhead. In what follows, we assume that each processor will have a continuation facility that independently assesses damage and effects whatever local changes are necessary for recovery in that processor.

Continued service after one or more processor failures requires that the following actions be performed:

(1) *Detect Failure*

Processor failure must be detected and communicated to the software on each of the remaining processors.

(2) *Assess Damage*

It must be known what processes were running on the failed processor, and what processes and processors remain. Further, processes executing on processors that survive the failure may still be affected by the failure. For example, their execution may depend on processes or contexts that were lost with the failed processor. If anything is to be done about such

processes, they must be known and there must be some way of communicating with them.

(3) *Select A Response*

Information must be provided so that a sensible choice of a response can be made. This choice will normally depend on which processors and processes remain, but in many applications the choice will also depend on other variables and their values would have to be known. The height of an aircraft, for example, might determine what actions should be taken when part of the avionics system is lost.

After a failure is reported to the software on a particular processor, the local reconfiguration software will independently decide on a response and put into effect the changes required on that processor. The choice of a response depends on the reconfiguration strategy provided by the programmer and on the information provided. It is important that the information which the reconfiguration strategy uses be consistent across processors, since if it is not, reconfiguration processes on different processors could decide on different responses and work at cross–purposes.

(4) *Effect The Response*

Once a response has been decided on, it must be possible to carry it out. The reconfiguration software should be able to create and remove processes, start and stop processes, and be able to communicate with processes so that they can take appropriate action on their own. In many cases the new processes will have to be provided with data, and a consistent set of such data would have to be available to the reconfiguration software.

Various difficulties are raised by these requirements. Firstly, they depend strongly on data that is consistent across all machines. Without making quite unrealistic assumptions about the underlying message passing system, it cannot be assumed that data is consistent when a processor fails if no precautions are taken. However, a two phase protocol [9,10] can be used in this situation to ensure that consistent data (but perhaps not the most recent values) is available on all machines.

A second problem with the view of continuation taken above is the treatment of unrecoverable objects [11]. If an unrecoverable object has been modified, backward error recovery is not possible following a failure. The problem is no different on a distributed system than on a uniprocessor system. An apparent difference is that all the processors in a distributed system need to be informed of changes to unrecoverable objects and this has to be done in the presence of failures. However, distribution of status information about unrecoverable objects is just an example of the data consistency problem discussed above.

V DISTRIBUTION AND CONTINUATION IN Ada

We now consider the use of Ada for programming distributed systems in which processor failure has to be tolerated. What is needed is a distributed system that provides the continuation facilities discussed above. None of these facilities are explicitly provided in Ada and very few can be realized using existing features of the language. These difficulties are discussed in this section. We show in later sections how the difficulties can be overcome by making minor additions to the semantics of Ada; no changes to the syntax are necessary.

Distribution

The choice of objects to be distributed is an important question in the design of a distributed system. Ada has a tasking mechanism and, according to the Ada Reference Manual [2], it is intended that tasks be distributed in an Ada program:

> Parallel tasks (parallel logical processors) may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor.

Also, it is clear from the requirements for the language [12] and from the Ada Reference Manual [2] that the tasking facilities are intended to be used for all task communication and synchronization even when different physical processors are executing the tasks involved.

It would be possible to devise a separate mechanism for inter–task activities between computers using some form of input and output. However, this would be equivalent to communication by shared memory and would be less secure than the existing language facilities because compile–time checking would not be possible. Further, it would be specific to the program for which it was written, and a program design change that required a task to be relocated to a different computer could force substantial changes in those tasks that communicate with it.

No facilities are defined in Ada to control the distribution of tasks and this leads to various difficulties. For example, it is essential that software to be used following the failure of a particular processor not be resident in the memory of that processor (otherwise the system would be a centralized). To achieve this separation, the programmer must control the placement of both the primary and alternate software. Surprisingly, although there are representation clauses to control the bit–level layout of records, to allow the placement of objects at particular addresses within a memory, and to associate interrupt handlers with specific machine addresses, there is no explicit mechanism for control of distribution in Ada.

It is not sufficient to be able to control the allocation of tasks to processors. There must be explicit semantics for task distribution and these semantics *must* deal with processor failures. For example, if there are multiple tasks of a particular task type and they are executing on different processors, a separate copy of the code must be required for each processor. Otherwise, an implementation could provide a single copy of the code that was shared by all processors; for example by fetching a copy when a new task body is elaborated. This would be satisfactory if there were no failures. However, failure of the processor containing the original copy of the code would then suspend all subsequent elaborations.

Although Ada was designed for distributed systems, neither the syntax nor the semantics for task distribution is defined.

Continuation

For any particular programming language, the required continuation facilities discussed in section IV could be provided in three different ways:

(1) By using mechanisms in the programming language specifically designed for that purpose.

(2) By using mechanisms in the programming language that were designed for another purpose. If this were done, it would be a coincidence if the mechanisms worked satisfactorily since they were not designed to support fault tolerance.

(3) By using mechanisms outside the programming language such as modifications to the execution–time environment or software written in some other language, perhaps an assembly language.

Unfortunately, Ada makes no explicit provision for continuation. Many features of the language raise substantial difficulties in damage assessment, and in selecting and effecting a response.

Failure Detection

An execution–support system for Ada is not required to provide any facilities for detection of processor failure. No specific interface is provided by the language to allow software to be informed of processor failure.

If failure could be detected, it might be possible to inform the software by raising an exception or generating an interrupt; the latter using an entry call as its interface. In either case, appropriate placement of the corresponding exception handler or **accept** statement becomes a problem since it must be assured that they will be executed when required. Also, the necessary exception and entry names are not predefined and so their use is neither standardized nor required.

Damage Assessment

Clearly, the damage sustained as a result of a processor failure includes loss of the services that were provided by the software that was executing on the processor that failed. It also includes loss of the data contained in the memory of the failed processor. In addition, in an Ada program the failure of a processor can cause damage to the software that remains. Broadly

speaking, two forms of damage can occur. A task can be suspended waiting for a message that will never arrive, and a task can lose part of its context. These will be discussed in turn.

Task Communication

The problems that arise in task communication are best illustrated by an example. Consider an Ada program that contains two tasks A and B executing on different processors. Suppose that task A has made a call to an entry in task B, and that B has started the corresponding rendezvous. If B's processor now fails, task A will remain suspended forever because the rendezvous will never end. Since the failure takes place after the start of the rendezvous, a timed or conditional entry call will not avoid the difficulty.

Similar problems arise throughout Ada, both in explicit communication such as the rendezvous and in implicit communication such as task activation. A detailed examination of these situations is given in appendix 2 and in [13].

Loss Of Context

In block structured languages, a program unit can assume the existence of an instance of all objects in the surrounding lexical blocks. When a system is distributed, it is possible to have a given program unit on one processor and one of its surrounding lexical blocks on another processor. If the latter processor fails, it must be decided what to do with the surviving inner program unit.

A task in Ada relies on several contexts; the context of the body, the creator, and the masters [2]. All of these contexts may be different. Each of them may be lost due to processor failure. Ada defines no semantics for these situations.

In summary, the damage following processor failure will include lost services, lost data, the permanent suspension of tasks on remaining processors for a variety of reasons, and the loss of contexts of some tasks. This damage could be quite extensive. As presently defined, Ada provides no way of determining the extent of this damage.

Selecting and Effecting The Response

The purpose of effecting a response is to provide alternative services for the services that were lost. The source of the new services will have to be software that resides on machines that remain after the failure.

Ada has facilities for starting tasks and for creating them. It is a relatively simple matter to cause software to begin execution to replace lost services. Note however that the replacement software is part of the same program as the software that was lost, and so cannot use names that would be ambiguous to a compiler. Thus for example a replacement task cannot be given the same name as the task it is intended to replace. Although tasks can be created and started, the scope rules of the language limit the lexical placement of the alternate software since it must be within the scope of the software effecting the response. Recall also that Ada does not provide the necessary control over distribution.

Even if replacement services could be started and distribution could be controlled, it is still necessary for the replacement software to communicate with the software remaining after the failure. This means that the communication paths used before the failure have to be redirected. Communication will be primarily by the rendezvous. The rendezvous in Ada is asymmetric; a calling task needs to know the name of any task containing an entry it wishes to call, but a called task need not know the names of tasks that will call it. If a calling task has to be replaced because of a failure, the replacement can call the same entry that was called by the lost task. The entry is still available in the same task that was being called before the failure. Thus redirection is trivial if a calling task is lost.

However, if a called task has to be replaced because of a failure, the replacement cannot be given the same name as the task that was lost. This would duplicate the definition of a task name in the same scope. Thus, in this case, redirection is quite involved. The replacement called task will have to have a different name and, more importantly, all the calling tasks (that may not have been replaced) will have to begin using a different name in their entry calls.

This difficulty is not quite as serious for tasks created by allocators. Since assignment is allowed for access variables, communication can be redirected by assigning a value representing an alternate task to an access variable used to make entry calls. Two problems then arise. First, the entire program has to be written using access variables to reference tasks. Second, an alternate task has to be of the same type as the primary task and this makes the provision of alternate service difficult.

Selecting and implementing a response relies on the availability of data that is consistent across machines. Ada makes no provision for ensuring that data can be reliably distributed across machines.

Finally, in effecting a response, it will be necessary to take care of those tasks damaged by the failure. The only way that this can be done in Ada is to abort them. Further, since some computing facilities have been lost, the response that is chosen might also involve modifying services that were not affected by the failure by aborting some tasks and starting others. Because of the scope rules of the language, aborting several tasks will be difficult to arrange if the program makes use of nesting and a single piece of software is to contain all the necessary abort statements.

VI FAILURE SEMANTICS FOR Ada

It is possible to overcome the difficulties discussed above. The first step is to fill in the gaps in the semantics of Ada. In particular, the meaning of distribution and the effect of damage to the remaining software caused by a processor failure must be specified. While it is not difficult to choose a reasonable meaning for distribution, the problem of what to do with damaged tasks is much more difficult. It must be emphasized that the semantics suggested in this section were chosen so as to follow the existing semantics as much as possible.

Distribution

The primary aim of distribution semantics is to avoid the possibility of a centralized distributed system. It will be assumed that only tasks will be distributed. The distribution of a task T to a processor P will be taken to mean that the task activation record for T and all of the code for T will be resident on P.

Damaged Tasks

In section IV it was pointed out that the failure of a processor may affect tasks running on the remaining processors, and that many language features can cause these problems. The difficulties do not arise because tasks were lost when the processor failed. Any task could be removed from an Ada program at essentially any point *without* processor failure by execution of an **abort** statement. Rather, the difficulties arise because the semantics of the language fail to deal with the situation. Ada semantics *are* precisely defined for tasks being aborted and for the subsequent effects on other tasks, and the execution–time system is required to cope with the situation. We suggest therefore that the damage following processor failure be handled as if the tasks that were lost had been aborted. This would allow the language–dependent part of the damage following processor failure to be treated using existing language facilities.

This choice of semantics leads to a new problem that needs to be addressed; the status of the main program following failure. By definition all non–library tasks in an Ada program are nested inside the main program and so depend upon it. If failure of a processor is to be treated as if **abort** statements had been executed on the lost tasks, then a serious problem arises with the main program. When a task is aborted, all its dependents are aborted also. For any task lost through failure this is reasonable. It means that all the dependents that were not lost with the task have to be aborted by the system. However, if the main program is lost, this implies that all the tasks that depend on the main program (that is almost all of them) will have to be aborted. This effectively removes the entire program. Clearly this is unsatisfactory. We suggest therefore that the main program has to be treated as a special case. For the main program, and only for the main program,

the execution–support system will have to create an exact replacement if the main program is lost through processor failure. To ease the overhead that this involves, we suggest that the sequence of statements in the **begin end** part of the main program be limited to a single **null** statement. The main program should be used only to define global objects and tasks that will manipulate these objects.

VII A SUPPORT SYSTEM STRUCTURE FOR Ada

Although Ada does not support the facilities required for continuation explicitly, the semantics described in section VI can be achieved if the execution–time support structure is suitably modified. In this section we discuss the necessary modifications and in section VIII we show how they are used with Ada.

Failure Detection

Failure detection could be performed by hardware facilities over and above those provided for normal system operation. Alternatively, failure could be detected by system software. The hardware option is less desirable because it requires additions to existing or planned systems, and the detection hardware itself could fail. We suggest therefore the use of software failure detection.

Software failure detection can be either passive or active. A passive system might rely on tasks assuming that failure had occurred if actions did not take place within a "reasonable" period of time i.e. timing out. Alternatively, a passive system could require that all messages passed between tasks on separate processors be routinely acknowledged. This is a particularly simple case of timing out since failure has to be assumed if no acknowledgement is received.

The disadvantages of passive detection are:

(1) Timing out assumes an agreed–upon upper limit for response time.

(2) A failed processor will not be detected until communication is attempted and this may be long after the failure has occurred.

Upper bounds on response time may be hard to determine. Very complex situations can arise from an incorrect choice. The reason for a lack of response from a task on another processor may not be failure of that processor but merely a temporary rise in its workload. The consequences of timing out could be an assumption by one processor that another had failed, followed by reconfiguration to cope with the loss. Clearly, if this assumption is wrong, two processors could begin trying to provide the *same* service.

Being unaware that a processor had failed will lead to a loss of the service it was providing until the failure is noticed. In a system with many processors each providing relatively few services, the amount of inter–processor communication might be quite low. A failed processor may then go unnoticed for so long that damage to the system being controlled might result.

It is for these reasons that we reject passive software failure detection and suggest the use of active software failure detection. In an active system, some kind of inter–processor activity is required periodically and if it ceases, failure is assumed. The messages that are passed are usually referred to as *heartbeats*. Multiple failures may occur at essentially the same time but transmission times may vary. Since it is important that machines surviving failure have a consistent view of the system state before they begin reconfiguration, the heartbeats must be organized so that each remaining processor gets the same information about the failure. There are many ways to achieve this. For example, all machines may be required to generate their heartbeats at approximately the same time so that each machine will receive all the heartbeats of the other machines in a given interval. Any not received in this interval can be assumed to have failed.

A final question of implementation is whether the generation and monitoring of heartbeats should be the responsibility of the programmer or the Ada execution–time support system. We favor the execution–time support system for reasons discussed below.

Damage Assessment

The mechanism that we propose to cope with this, together with the heartbeat mechanism, is shown in figure 2.

Whenever any communication takes place between tasks on different processors, the execution–time support system on the processor starting the communication records the details in a *message log*. Whenever a failure is detected, each processor checks its message log to see if any of its tasks would be damaged by the failure (permanently suspended for example). If any are found, they are sent fake messages. They are called "fake" because they are constructed to appear to come from the failed processor but clearly do not. The message content is usually equivalent to that which would be received if the lost task had been aborted. In this way, each processor is able to

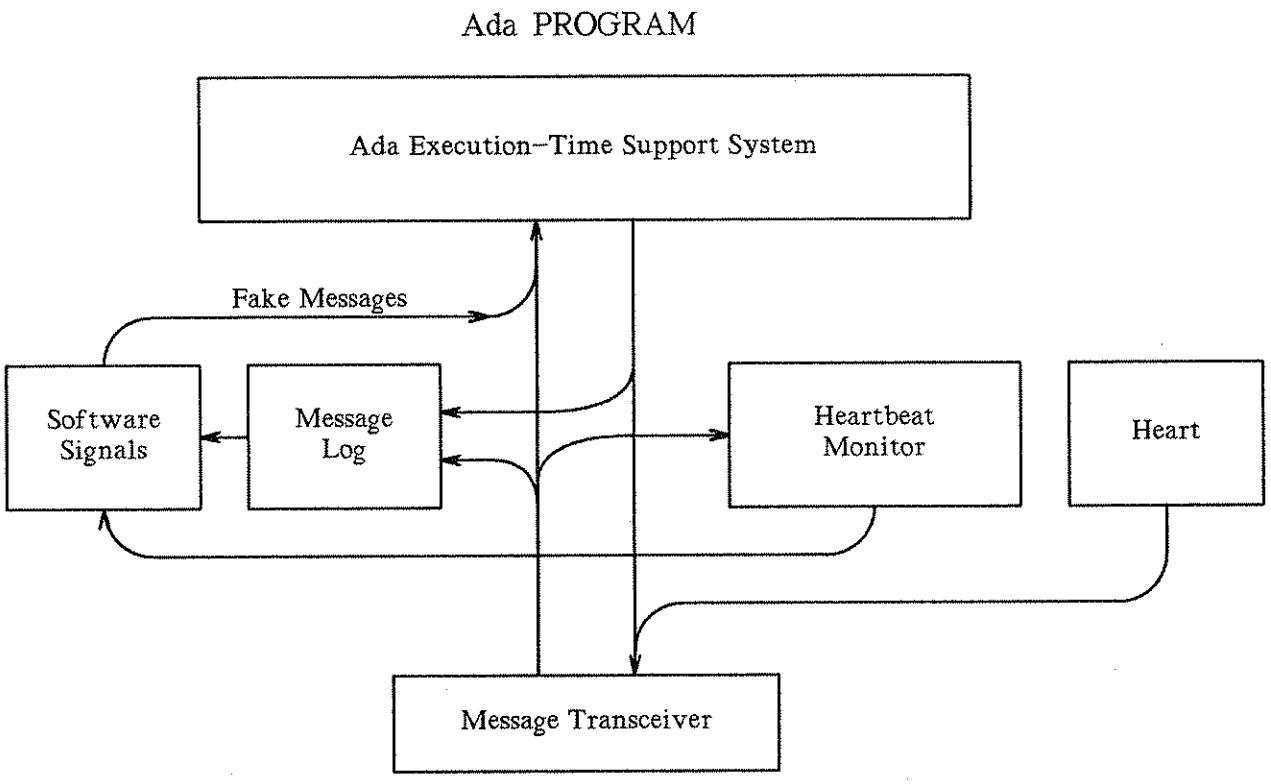

Fig. 2. Implementation Model For Each Processor

ensure that none of its tasks is permanently damaged, and the action following failure for each remaining task is that which is associated with an abort. It often takes the form of raising an exception.

Clearly it is possible for unsuspecting tasks to attempt to rendezvous with tasks on the failed processor after failure has been detected, signaled, and other communications terminated. This situation can be dealt with easily if the execution–time support system returns a fake message immediately indicating that the serving task has been aborted and that rendezvous is not possible.

Because a fairly extensive set of facilities is required in the execution–time system for fake messages, we suggest that the heartbeats be handled here also. There is a clear need for cooperation between the heartbeat monitoring system and the fake message system. Operating both at the same level is probably the only practical approach. This has the additional advantage that the programmer is not burdened with the need to include the heartbeat system in his program. Finally, the heartbeat system is so central to the reliability of the entire system that it should operate at the lowest practical level of the software system. Thus it relies for its operation on the correct operation of the minimum amount of other software.

Selecting and Effecting The Response

Since consistent data across machines is essential to allow a response to be chosen and implemented, the execution–time support system for Ada must provide a mechanism for ensuring that data can be distributed reliably. As mentioned in section IV, a two phase protocol can be used and we propose that an implementation of it be included in the execution–time system along with the message log and heartbeat mechanism.

VIII FAULT TOLERANCE IN Ada

In this section we show how a fault–tolerant Ada system can be built using the support mechanisms just described. As was shown in section V, Ada does not provide any specific facilities

to support this type of fault tolerance. Existing features of the language that were not designed for the purpose have to be used to interface with the modified support system.

Failure Detection

When failure of a processor is detected by the heartbeat mechanism, this information must be transmitted to the software running on each remaining processor so that reconfiguration can take place. The information is available to the execution-time support software in some internal format, but it has to be transmitted to the Ada software using an existing feature of the language.

As noted in section V, if the language is not to be changed, one approach is to make use of the language's exception mechanism and have the execution-time system generate a predefined exception on each processor remaining after a failure. If this is done, it is not clear where the handler for the exception should be placed. The handler will be receiving extremely important information (namely that a failure has occurred) and, in order to deal with the situation, it must be guaranteed that the handler will be executed. Unless handlers for the exception are placed in every task that might be running, execution of the handler cannot be guaranteed. An alternative is to define a task to contain the handler and to raise the exception in this task only. Clearly, this task should not be executing until the exception is raised. Unfortunately, there is no way for an exception to activate a task in Ada.

Another approach to signaling failure is to view the required signal as being like an interrupt, and transmit the information to the Ada software by a call to a predefined entry. Again there is the problem of where the entry should be defined to ensure execution. However, in this case, the solution of defining a task and defining the entry within it works very well. If the task is given the highest priority, it will be suspended on the entry until the call that signals failure, whereupon it will immediately begin execution.

We propose therefore that a task (RECONFIGURE_I) containing an entry with a single parameter be defined on each processor (I is the processor number). The **accept** statement associated

with the entry will be in an infinite loop. This task will normally be suspended on the **accept** statement for the entry and, when a failure is detected by the heartbeat mechanism, a call to the entry will be generated. The parameter passed will designate which element of the system's hardware has failed. The task will then be activated and will contain code within the **accept** statement to handle reconfiguration. A general form of the body of this task is shown in figure 3. Since this task is in an infinite loop, it returns to the **accept** statement once a particular failure has been dealt with. Thus subsequent failures will be dealt with in the same way and, in principle, any number of sequential failures can be dealt with. Further, if physical damage removes more than one processor at the same time, the remaining processors will notice the loss of heartbeats in some order, and calls to the entry in the reconfiguration task will be generated sequentially. Thus multiple failures occurring together will be dealt with as if they had occurred in some sequence.

Damage Assessment

Given the support system described in section VII, damage will be limited to lost tasks and lost data. No remaining tasks will be suspended. Each task that could have been suspended will have received fake messages giving it the impression that the task with which it was communicating had been aborted. It is the programmer's responsibility to ensure that the

```
task body RECONFIGURE_I is
begin

  -- Initialization code.
  loop
    accept FAILURE(WHICH : in PROCESSOR) do
      -- Code to handle hardware failures.
    end accept;
  end loop;

end RECONFIGURE_I;
```

Fig. 3. Body Of Task RECONFIGURE_I.

subsequent actions by these tasks are appropriate. In addition, all tasks losing contexts will have been aborted.

The tasks and data that were lost need to be determined. Provided there is control of distribution (see below), this is quite simple. The information about which tasks are on which processors could be maintained and referenced in three ways:

(1) It could be stored in a table within the program itself.

(2) It could be stored within a table maintained by the execution–support system. This table will be needed in any case because it is required to implement inter–task communication. Provision could be made for the program to interrogate it.

(3) The information could be stored implicitly within the program. If all task–to–processor assignments are known at compile time and do not change, the code used for reconfiguration following failure can be written with the distribution information as an assumption.

There is no clear advantage to any of these methods. The choice in any particular case is implementation dependent. In the example given in appendix 1 we use the third method.

Selecting and Effecting The Response

Algorithms for the selection of a suitable response, and the algorithms used in that response, depend for their correct operation on having appropriate data available. Typically, each piece of data being manipulated by a program for an embedded application can be regarded as either *expendable* or *essential.* For example, partial computations and sensor readings would be expendable whereas navigation information and the status of weapons would be essential.

Expendable data need not be preserved across machine failures. A partial computation, for example, is only of value to the expression generating it. Replacement software that will be used following a failure can simply recompute any expendable values. A sensor value for example, is usually only useful for a short time and a suitable replacement value can be obtained by reading the sensor again. We suggest that any data items that the programmer considers expendable be given no special attention, and that the replacement software be written with the assumption that these data items are not available.

Essential data does need to be maintained across machine failures. In an Ada program this could be implemented in two ways. First, data items that the programmer considered to be essential could be marked as such (perhaps by a pragma), and the system would then be required to ensure that copies of these data items were maintained on all machines. Each time the data item was modified, all the copies would be updated. In the event of failure, one of the backup copies could be used immediately. This is simple for the programmer but potentially inefficient. Consider for example a large array that was designated as essential. If it were being updated in a loop, as each element was changed, it would be necessary to update all the copies of that element. The entire overhead associated with maintaining consistent copies would be incurred for each element change. In practice in order to allow reconfiguration, it would probably be adequate to wait until all the elements of the array had been modified and then update all the copies of the array at once.

A second approach is to provide the programmer with the tools to generate consistent copies across machines. In this way, not only the data items to be preserved but also the times during execution when copies will be made will be under the programmer's control. We suggest that this could be done by providing a data management task (DATA_CONTROL_I) on each processor. The data management tasks would together maintain a consistent set of essential data across all processors. When a task had produced some essential data, it would pass that data to the local data management task by calling the entry DATA_IN. The data management task would cause the necessary copies to be made and distributed while the calling task waited. Completion of the rendezvous would indicate that the distribution had been satisfactorily completed. When essential data was required (typically for the initialization of an alternate task) the task requiring it would obtain it from the local data management task by calling another entry DATA_OUT. A general form of the body of this task is shown in figure 4.

Although Ada has no specific facilities for control of task distribution, limited control can be achieved using either an implementation–dependent pragma or an implementation–dependent address clause. The pragma could have a machine notation as a parameter and be required to appear in the specification of the task or task type to which it applies, as is done with the predefined pragma **priority.** The distribution pragma would require the compiler to generate instructions and

```
task body DATA_CONTROL_I is
begin
  loop
    select
      accept DATA_IN(DATA : in DATA_KIND);
        -- Receive data from local tasks, distribute data to
        -- other data management tasks using two-phase commit.
    or
      accept DATA_OUT(DATA : out DATA_KIND);
        -- Provide data to local tasks.
    end select;
  end loop;
end DATA_CONTROL_I;
```

Fig. 4. Body Of Task DATA_CONTROL_I.

loader directives for the designated task to ensure that it is placed in the required machine. This is the notation used in the example given in appendix 1. Alternatively, for tasks created by allocators, the distribution pragma could be required to appear in a declarative part and it would apply only to the block or body enclosing the declarative part (similar to the predefined pragma **optimize).** Its effect would be to cause all tasks created by allocators in the block or body to be distributed to the machine designated by the pragma.

Address clauses could be used similarly. For example, in a particular implementation, the identifier parameter in the address clause could be interpreted as a task name, and the expression parameter as a machine designation.

As was pointed out in section V, the creation and deletion of tasks that might be required as part of effecting a response is easily achieved in Ada using allocators and the **abort** statement. Thus the particular **accept** statement within the reconfiguration task that is executed for a given failure can create and delete whatever tasks are needed to provide alternate service. A simpler approach to providing replacement software is to arrange for the required replacement task to be present and executing before the failure, but suspended on an entry. Such a task would not consume any processing resources although it would use memory, but it could be started by the reconfiguration software very quickly and easily by calling the entry upon which the replacement

task is suspended. A general form for a replacement task is shown in figure 5. This is the mechanism used in the example in appendix 1.

Redirection of communication to alternate software that has been started following a failure has to be programmed ahead of time into tasks that call entries in tasks that might fail. It will be necessary for the reconfiguration task on each processor to make status information about the system available to all tasks on that processor. Each task must be prepared to deal with a TASKING_ERROR exception after making a call to an entry on a remote machine in case the entry has changed because of failure.

In summary, an Ada program that uses the support system described in section VII to allow it to tolerate the loss of one or more processors would have the following form:

(1) A main program with a **begin end** section consisting of a single **null** statement.

(2) A static structure in which there is little or no nesting of the application and alternate tasks themselves. They may define nested tasks for their own use. This is necessary to ensure that these tasks are visible to the reconfiguration and data consistency tasks.

(3) A set of tasks providing the various application services; the distribution of the tasks being controlled by an implementation-defined pragma or address clause. Each task would contain

```
task ALTERNATE_SERVICE is
  pragma distribute(PROCESSOR_I);
end ALTERNATE SERVICE;

task body ALTERNATE_SERVICE is
begin
  -- Code necessary to initialize this alternate service.
  accept ABNORMAL_START;
    -- This task will be suspended on this entry until it is
    -- called by RECONFIGURE_I following failure. The
    -- code following the accept statement provides the
    -- alternate service. Any data required can be
    -- obtained from DATA_CONTROL_I.
end ALTERNATE_SERVICE;
```

Fig. 5. General Form Of A Replacement Task.

handlers for exceptions (such as TASKING_ERROR) that might be generated by the support system if failure occurred while that task was engaged in communication with a task on a remote machine.

(4) A set of tasks designed to provide any alternate service that the programmer chooses; each alternate task suspended on an **accept** statement that will be called to start it executing.

(5) A task on each processor designed to cope with reconfiguration on that processor; this task containing an entry with a parameter that indicates which hardware component has failed. These entries would be called automatically be the support system following failure detection.

(6) A task on each processor designed to distribute copies of essential data for tasks on that processor. Rendezvous with this task allows any other task to distribute essential data at any time the programmer chooses.

IX CONCLUSION

Although the probability of failure per unit time for a modern fault-tolerant processor is low, it is not zero. The loss of processors in a distributed system is certain to occur and must be anticipated. In order to benefit from the flexibility of distributed processing, crucial systems must be able to deal with processor failures.

Ada was designed for the programming of embedded systems, many of which are crucial and distributed. We have examined Ada's suitability for programming distributed systems in which processor failure has to be tolerated and found it to be inadequate. The difficulties have been discussed and proposals to avoid them have been suggested. These proposals involve additions to Ada's semantics, extensive modification to the execution-time system used by Ada, and careful organization of the Ada program itself.

Although the discussion presented here is in terms of distributed systems, similar problems can arise in shared–memory multiprocessor systems where processor failure has to be anticipated. If the system is organized so that different processors execute different tasks, processor failure at an arbitrary point could produce the damage discussed in section V.

Since we consider the non–transparent approach to be the only one that is feasible, the fact that Ada makes no explicit provision for this type of fault tolerance is unfortunate. The solution presented in this paper uses existing features of the language and is far from ideal. Modifications to Ada would be preferable.

An implementation of the tasking and exception handling features of Ada incorporating the ideas described in this paper is being undertaken. It is presently being tested.

APPENDIX 1

A Programming Example

This is a very simple example designed to illustrate some of the ideas discussed in this paper. In a typical Ada application, the program would be much larger and would have to take into account all the language features mentioned.

The example consists of a calling task CALLER that operates on one processor (CPU_1) and a serving task SERVER that operates on another processor (CPU_2). The calling task does some real-time processing and calls an entry in the serving task in order to get some form of service. The program is written to cope with failure of either processor. Alternates are provided for the calling and the serving tasks, and both a reconfiguration task and a data management task are present on each processor.

Normally only the calling and serving tasks are executing and a *fault-intolerant* version of this example would consist of just these two tasks. If processor one fails then it is necessary to start an alternate calling task on processor two. Similarly, if processor two fails it is necessary to start an alternate serving task on processor one.

The alternates are present on the required machines when the program starts execution. Each alternate is waiting on an entry named ABNORMAL_START so that they do no processing while both processors are operational. When one processor fails, the run-time system generates an entry call on the other processor to an entry in its task RECONFIGURE_I (where I is the processor number). This task then calls the ABNORMAL_START entry for the alternate which will obtain any data needed from the local data management task, and processing is able to continue.

If a rendezvous is in progress when the failure occurs and the calling task has been lost, then the serving task can complete the rendezvous. If, on the other hand, the serving task has been lost the calling task will be indefinitely suspended. In this case therefore, TASKING_ERROR is raised by the run-time system in the calling task. This frees the calling task and allows it to prepare

itself to use the alternate server.

If a rendezvous is not in progress when the failure occurs then processing on the remaining processor can continue. If this processor is executing the caller, then the caller will receive TASKING_ERROR the next time it attempts to rendezvous with the server and it will be reconfigured at that time. The alternate server will have already been started by then.

In this example, a single array called DATA is regarded as essential and the programmer has decided that it needs to be made consistent across both machines only rarely. The entry DATA_IN is called in tasks DATA_CONTROL_1 and DATA_CONTROL_2 periodically to achieve this.

Distribution is controlled in this example by the pragma **distribute_to** that takes a processor name as its parameter. It must appear in the specification of the task to which it applies, and is much like the predefined pragma **priority.**

```
procedure EXAMPLE is

  type PROCESSOR is (CPU_1, CPU_2);

  type DATA_KIND is array(1..100) of integer;

  task CALLER is
    pragma distribute_to(CPU_1);
    pragma priority(1);
  end CALLER;

  task body CALLER is
    type STATE  is (NORMAL, ABNORMAL);
    SYSTEM_STATE : STATE := NORMAL;
    DATA         : DATA_KIND;
  begin
    -- Initialization.
    loop
      MAIN_BLOCK:
      begin
        case SYSTEM_STATE is
          when NORMAL =>
            -- Normal pre-rendezvous processing.
            SERVER.E;
            -- Normal post-rendezvous processing.
          when ABNORMAL =>
            -- Abnormal pre-rendezvous processing.
            ALTERNATE_SERVER.E;
            -- Abnormal post-rendezvous processing.
        end case;
      exception
        when TASKING_ERROR=>
          SYSTEM_STATE := ABNORMAL;
      end MAIN_BLOCK;
      DATA_CONTROL.DATA_IN(DATA);
    end loop;
  end CALLER;

  task ALTERNATE_SERVER is
    entry ABNORMAL_START;
    entry E;
    pragma distribute_to(CPU_1);
    pragma priority(1);
  end ALTERNATE_SERVER;
```

```
task body ALTERNATE_SERVER is
begin
  -- Initialization.
  accept ABNORMAL_START;
  loop
    -- Alternate pre-rendezvous processing.
    accept E;
    -- Alternate post-rendezvous processing.
  end loop;
end ALTERNATE_SERVER;

task RECONFIGURE_1 is
  entry FAILURE(WHICH : in PROCESSOR);
  pragma distribute_to(CPU_1);
  pragma priority(2);
end RECONFIGURE_1;

task body RECONFIGURE_1 is
begin
  loop
    -- System calls the following entry when CPU 2 fails.
    accept FAILURE(WHICH : in PROCESSOR) do
      -- Start alternate server on CPU 1.
      ALTERNATE_SERVER.ABNORMAL_START;
    end accept;
  end loop;
end RECONFIGURE_1;

task DATA_CONTROL_1 is
  entry DATA_IN (VALUES : in  DATA_KIND);
  entry DATA_OUT(VALUES : out DATA_KIND);
end DATA_CONTROL_1;

task body DATA_CONTROL_1 is separate;

task ALTERNATE_CALLER is
  entry ABNORMAL_START;
  pragma distribute_to(CPU_2);
  pragma priority(1);
end ALTERNATE_CALLER;

task body ALTERNATE_CALLER is
  DATA : DATA_KIND;
begin
  -- Initialization.
  accept ABNORMAL_START;
  DATA_CONTROL_2.DATA_OUT(DATA);
  loop
    -- Alternate pre-rendezvous processing.
    SERVER.E;
    -- Alternate post-rendezvous processing.
  end loop;
end ALTERNATE_CALLER;
```

```
  task SERVER is
    entry E;
    pragma distribute_to(CPU_2);
    pragma priority(1);
  end SERVER;

  task body SERVER is
  begin
    -- Initialization.
    loop
      -- Pre-rendezvous processing.
      accept E;
      -- Post-rendezvous processing.
    end loop;
  end SERVER;

  task RECONFIGURE_2 is
    entry FAILURE(WHICH : in PROCESSOR);
    pragma distribute_to(CPU_2);
    pragma priority(2);
  end RECONFIGURE_2;

  task body RECONFIGURE_2 is
  begin
    -- System calls the following entry when CPU 1 fails.
    accept FAILURE(WHICH : in PROCESSOR) do
      -- Start alternate server on CPU 1.
      ALTERNATE_CALLER.ABNORMAL_START;
    end accept;
  end RECONFIGURE_2;

  task DATA_CONTROL_2 is
    entry DATA_IN (VALUES : in  DATA_KIND);
    entry DATA_OUT(VALUES : out DATA_KIND);
  end DATA_CONTROL_2;

  task body DATA_CONTROL_2 is separate;


begin
  null;
end;
```

APPENDIX 2

Damage In Task Communication

In Ada, there are many ways that damage can result when processor failure occurs during communication. The simple rendezvous will be examined in detail as an example. Other language elements involving task communication are considered only briefly since, for the most part, the difficulties that arise from processor failure are similar to those that arise in the simple rendezvous. The phrase "the processor executing task X fails" will be abbreviated to "task X fails" whenever no confusion arises.

A simple rendezvous in Ada consists of a calling task C making an entry call, S.E, to a serving task S, that contains an **accept** statement for the entry E. The syntax is shown informally in figure 6. The semantics of the language require that if the call is made by C before the **accept** statement is reached by S, C is suspended until the **accept** statement is reached. If S reaches the **accept** statement before the call is made by C, S is suspended until the call is made. In either case, C remains suspended until the rendezvous itself is complete.

In order to look at the effects of processor failure on the rendezvous, it is necessary to specify an implementation at the message passing level. Only the simple case of a task C calling an entry E in a serving task S will be considered. Further, we assume that the call is made before S has reached the corresponding **accept** statement; the case where the serving task waits at its **accept**

```
Calling Task C                    Serving Task S

      .                             ACCEPT E DO
      .                                 .
     S.E;                               .
      .                                 .
      .                             END E;
```

Fig. 6. The Syntax Of A Simple Rendezvous.

statement is similar. One possible message sequence is shown in figure 7.

The calling task C asks to be put onto the queue for entry E. When S reaches its **accept** statement for E, it sees that C is on the queue. C can be considered to be engaged in the rendezvous after the RENDEZVOUS_START message arrives at C. When the rendezvous is completed the RENDEZVOUS_COMPLETED message would awaken C which would continue. We assume that all messages arrive safely.

Using this implementation of a simple entry call, what happens if either the serving or the calling task fails? A detailed examination of all possible cases has appeared elsewhere [13] and will not be repeated here. However, it is clear that there are several situations in which processor failure could cause one of the tasks to be trapped. For example, if task S fails at any point after the RENDEZVOUS_START message is sent but before the RENDEZVOUS_COMPLETE message is sent, the latter will *never* be sent. Task C has no way of distinguishing this situation from a long service time by task S, and so will wait forever. Although the processor executing C is still working, task C is permanently suspended by the loss of a different processor.

It might appear that the timed entry call solves some of the problems raised above but it does not. The semantics of the timed entry call appear to be quite straight-forward [2]:

> A timed entry call issues an entry call that is canceled if a rendezvous is not started within a given delay.

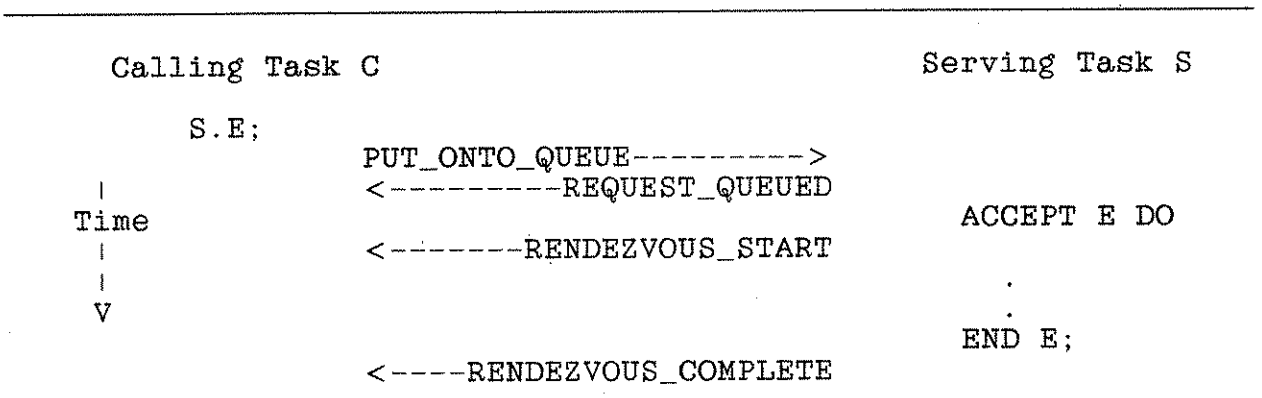

Fig. 7. The Messages Used To Implement The Rendezvous.

However, in a distributed system, messages will take time to get from a task on one processor to a task on another. Even if the underlying message passing system can guarantee that a message will eventually arrive correctly, this will be implemented at a lower level by a protocol that may well involve acknowledgement of messages, and the resending of messages that have been lost. A message can certainly be delayed for some arbitrary length of time. Even physical separation of the processors may impose a significant delay.

One possible interpretation of the timed entry call would be to count the total time until the rendezvous is started. Message passing time and time waiting on the entry queue would be included. This interpretation has to be ruled out by the statement in the language definition that a timed entry call with a delay of zero is the same as a conditional entry call:

> If a rendezvous can be started within the specified duration (or immediately, as for a conditional entry call, for a negative or zero delay), it is performed and the optional sequence of statements after the call is then executed.

If the delay included both message passing time and time on the queue, a delay of zero would be impossible and a timed entry call with a delay of zero would never succeed.

Another interpretation of the delay is that it is just the time spent waiting on the entry queue. We assume that this is the delay intended by the language definition since this has a meaning when the specified delay is zero.

A timed entry call gives protection against having to wait too long on the entry queue. Thus, it could be used to provide protection against processor failure *before* the rendezvous starts but not afterwards. An analysis of the message traffic necessary for the timed entry call can be performed that is similar to that shown in figure 7. The issues that arise when considering failure are similar but more extensive than those that arise with the simple rendezvous. What the task issuing the call needs is some guarantee that it will not be trapped in an attempt to communicate, and forced to miss a deadline. It does not matter to the task, whether the time is spent waiting on a queue, or attempting to send a message, or any other activity.

The conditional entry call is no more helpful than the timed entry call. Again, the semantics of the conditional entry call appear to be quite straight–forward [2]:

A conditional entry call issues an entry call that is then canceled if a rendezvous is not immediately possible.

By a similar argument to that used with timed entry calls, we conclude from the rules of the language that "immediately" must mean zero waiting time on the entry queue. As message passing time can vary, "immediately" may turn out to be an arbitrary delay. Apart from the semantic difficulties arising in a distributed system, the possibility of the caller being trapped indefinitely following processor failure occurs with conditional entry calls as with the other rendezvous.

We now consider the creation of nested tasks. Again, a detailed examination of the difficulties arising with task creation has appeared elsewhere [13]. Here, we give an example to show the potential problems. Task creation by allocators will not be considered; the difficulties that arise are similar.

A task is created in two steps. First, it is elaborated at which point entry calls can be made to it. Second, it is activated, that is, the declarative part of its body is elaborated and it begins execution. Elaboration of a task occurs as a part of the elaboration of the body of the declarative part of the parent unit. Activation of a task occurs after the elaboration of the declarative part of the parent unit. Conceptually, this occurs after the **begin** but before the first statement of the parent's body. The parent cannot be activated until all nested tasks have been activated.

To see the difficulties that task creation can raise, consider a set of three tasks P, A, and B, with A and B nested inside P. A and B have no other tasks nested within them and each task is to execute on a different processor; P on processor one, A on processor two, and B on processor three. The elaboration of the body of P includes the elaboration of A and B, and so messages will be sent from processor one to the other processors requesting the elaboration of A and B. Once this is done, tasks A and B can accept entry calls. When task P reaches its **begin,** all of the objects that it declares have been elaborated, and A and B are then activated. This requires an "activate A" message being sent from processor one to processor two and an "activate B" message being sent from processor one to processor three. The activation of P requires that the activation of A and B be complete, and so P cannot proceed until it has received responses to these activation messages indicating that A and B have been activated. Clearly there are numerous difficulties that can arise

if any one of the three processors fails. For example, P will be suspended forever if either processor two or three fails at any time before both A and B have completed activation. In that case, P could not proceed because one of its dependents would never be activated. Similarly, A would be trapped if it called an entry in B and processor three failed after B was elaborated but before it was activated. Both A and B would be trapped if processor one failed after the elaboration messages were sent to A and B, but before the activation messages were sent.

Task termination produces difficulties also. A task waiting at a **select** statement with an open terminate arm can be terminated if its master is completed and all other dependents of the master are either terminated or waiting at **select** statements with open terminate arms. In order to check this condition it is necessary to suspend all the dependents as they are checked. If this is not done, it is possible for two dependents each to be waiting at a **select** statement with an open terminate when checked, but never to both be waiting at **select** statements simultaneously. If the dependents are executing on a different processor from the master, it is necessary for the master to send messages to its dependents suspending them for the duration of the termination check. If the master task's processor fails before a message to resume is sent (assuming that termination is not possible), a suspended dependent will remain suspended forever.

Even accessing a non-local or a shared variable could cause the referencing task to be suspended. Access to a variable stored on a remote machine requires that a request message be sent and that a reply be received. Failure of the remote machine between the two messages would cause suspension of the requesting task. Although the language allows an implementation to use a copy of a shared variable, updating it only at synchronization points, and the definition of a synchronization point can vary depending on whether or not the variable is declared as **shared,** there are still implied update messages at synchronization points. Access to a shared variable on another processor requires that a dialogue take place, and failure of the processor on which the shared variable resides could trap the task attempting to reference the variable.

REFERENCES

(1) T. V. McTigue, "F/A-18 Software Development - A Case Study", Proceedings Of The AGARD Conference On Software For Avionics, The Hague, Netherlands, pp. 39.1-39.15, September 1982.

(2) Reference Manual For The Ada Programming Language, U. S. Department of Defense, 1983.

(3) J. H. Wensley, et al, "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", Proceedings of the IEEE, Vol. 66, pp. 1240-1254, October 1978.

(4) A. L. Hopkins, et al, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", Proceedings of the IEEE, Vol. 66, pp. 1221-1239, October 1978.

(5) A. S. Tanenbaum, "Network Protocols", ACM Computing Surveys, Vol. 13, pp. 453-489, December 1981.

(6) R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems", ACM Transactions On Computer Systems. Vol. 1, pp.222-238, August 1983.

(7) D. Cornhill, "A Survivable Distributed Computing System For Embedded Applications Programs Written In Ada", ACM Ada Letters, Vol. 3, pp. 79-87, December 1983.

(8) N. G. Leveson and P. R. Harvey, "Analyzing Software Safety", IEEE Transactions On Software Engineering, Vol. SE-9, pp. 569-579, September 1983.

(9) P. A. Alsberg and J. D. Day, "A Principle For Resilient Sharing Of Distributed Resources", Proceedings Of The International Conference On Software Engineering, San Francisco, October 1976.

(10) J. N. Gray, "Notes On Database Operating Systems", in Operating Systems: An Advanced Course, Springer-Verlag, New York 1978.

(11) P. A. Lee, "A Reconsideration Of The Recovery Block Scheme", Computer Journal, Vol. 21, pp. 306-310, November 1978.

(12) Department Of Defense Requirements For High-Order Computer Programming Languages - STEELMAN, U. S. Department of Defense, 1978.

(13) P. F. Reynolds, J. C. Knight, J. I. A. Urquhart, "The Implementation and Use of Ada On Distributed Systems With High Reliability Requirements", Final Report on NASA Grant No. NAG-1-260, NASA Langley Research Center, Hampton, Va.