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

John C. Knight
John I.A. Urquhart

Computer Science Report No. TR-86-19
November 10, 1986

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 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 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 tasks are the primary object of distribution, and that communication between tasks on separate processors will take place using the facilities of the Ada language. 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 and undesirable. In addition, the Ada language reference manual states specifically 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 that make its use on a distributed system very difficult if processor failures have to be tolerated. The issues are not raised from efforts to implement the language, but from the complete lack of semantics in Ada defining the state of a program when a processor is lost.

We define appropriate semantic enhancements to Ada, and describe the extensive modifications to the execution support required for Ada to implement these semantics. 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. If the required program structure is used and the necessary support system facilities are available, continued processing can be provided.

I INTRODUCTION

Ada [1] was designed for the programming of embedded systems, 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 facilities remaining after a hardware failure. This is important in *crucial* applications; applications such as defense or avionics systems where total failure of the computer system could have a very high cost.

By a distributed system we mean a set of processors linked together by a high-performance communications bus. Each processor would have its own memory, and devices such as displays, sensors, and actuators would be connected to the bus via dedicated microprocessors; these devices would be accessible from each processor.

Although much research has been undertaken in recent years to produce reliable computers [2, 3], these machines may still fail. Moreover, lightning, fire or other physical damage can cause a processor to fail no matter how carefully it is built. 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.

In this paper, 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 [4]. The kind of hardware failure that we are concerned with, the total loss of a processor with no warning, is not addressed by these protocols and must be dealt with separately.

We assume that the processors have the *fail stop* property [5]. This means a processor fails 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. 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. A complete break in the bus system that isolates some subset of the processors

(i.e. the network becomes partitioned) is very serious though extremely unlikely given multiple routes and replication. The issues that arise in this case are different from those arising from processor failure. They are beyond the scope of this paper and will not be dealt with here.

We begin in section II by looking at two approaches that might be taken to tolerating the loss of a processor and providing continuation. One approach relies on the support system providing a virtual target in which failures do not take place. In this approach, the application program is unaware of any failures that occur. In the second approach, the support system provides minimal facilities and the application program is expected to cope with the situation. The second approach is the subject of the remainder of the paper. In section III we consider the facilities that are required for continuation and discuss their relative absence in Ada. We show that the problem centers around the lack of semantics describing distribution and failure for an Ada program. Appropriate distribution and failure semantics for Ada based on the requirement that the language syntax remain unchanged are presented in section IV. An execution-time system to provide these semantics is presented in section V. We show how these semantics can be used in Ada programs in section VI and discuss their implementation and execution-time performance in section VII. Our conclusions are presented in section VIII.

II 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 failures. 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 refer to this approach as *transparent* since, in principle, the programmer is unaware of its existence. This is the approach to building fault-tolerant distributed Ada systems being pursued by Honeywell [6, 7].

Transparent continuation has several advantages. For example, the programmer need not be concerned with reconfiguration and need not know about the distribution. It follows that the same program can be run without modification on different systems with different distributions.

A disadvantage of transparent continuation is that the programmer cannot specify degraded or *safe* [8] service to be used following processor failure. Since such service cannot be specified by the system, transparent continuation must always provide full service or no service. In many crucial systems this is not acceptable. In practice, this approach may not even be permitted by agencies regulating crucial systems. Such agencies frequently require that the precise actions that will be taken following failure must be stated explicitly, and that degraded service must always be provided if any computing facilities exist after a failure.

A second disadvantage of transparent continuation is its overhead. 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.

This overhead will also be influenced by the program that is to be distributed. An unfortunate consequence of transparent continuation is that distribution and fault tolerance are not taken into account at the top level of design. Thus the program may be written in a way that makes distribution awkward and greatly increases the overhead.

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 which 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.

Non-transparent continuation has several disadvantages. For example, the programmer must be concerned with reconfiguration, and must either specify the distribution or be prepared to deal with any distribution provided by the system. Also, the program will depend on the system; at least the reconfiguration parts will.

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 has 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.

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.

III CONTINUATION REQUIREMENTS AND Ada

In the non-transparent approach, the programmer must deal with both distribution and continuation. 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, what processes and processors remain, and what state the processes that remain are in.

(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.

(4) *Effect The Response*
Once a response has been decided on, it must be possible to carry it out.

We note that these actions depend strongly on data that is consistent across all machines.

In this section we discuss, in detail, distribution, the actions listed above, and the provision of consistent data. It will be seen that many of the required continuation facilities are absent from Ada.

Distribution

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 centralized). To achieve this separation, the programmer *must* control the placement of both

the primary and replacement software.

It is not sufficient to be able to specify that the primary and replacement software be on different machines. It is essential that the programmer be able to designate exactly which machine each will reside on. If this is not possible, the reconfiguration software will have to be prepared to deal with the loss of any combination of the primary tasks when a machine fails.

It is not sufficient merely to be able to control the allocation of software to processors. There must be explicit semantics for distribution and these semantics *must* deal with processor failures. For example, in Ada 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.

The choice of objects to be distributed is an important question in the design of a distributed system. Thus a programming language that is to provide continuation must specify what units can be distributed, and exactly how the distribution is to be done. We refer to these as the *distribution semantics*. In addition, the language should include a syntax for expressing distribution.

Ada has a tasking mechanism and, according to the Ada Reference Manual [1], 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 [9] and from the Ada Reference Manual [1] 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 consider distributing complete programs [10]. This would require the development of 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. An excellent discussion of the disadvantages of distributing complete programs is given by Cornhill [7].

No facilities are defined in Ada to control the distribution of tasks. 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. Although Ada was designed for distributed systems, neither the syntax nor the semantics for task distribution is defined.

Failure Detection

Normally, a facility for failure detection would be provided by the underlying system. The application software will have to be informed of the failure so that it can take the necessary actions to continue. The programming language should provide an interface that would allow the failure to be signaled. No specific interface is provided by Ada to allow software to be informed of processor failure.

It is 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 is a problem since it must be assured that they will be executed promptly. 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, the failure of a processor can affect the software that remains. The programming language definition must specify exactly which part of the remaining software will be affected and exactly how it is affected. We term this the *failure semantics* of the language.

Ada does not define any failure semantics. However, it is clear from the rules of the language that a great deal of the remaining software can be affected by the loss of a processor. Broadly speaking, this can happen is two ways. A task can be suspended during communication waiting for a message that will never arrive, and a task can lose part of its context.

The problems that arise in task communication, the first way that a task can be affected by processor failure, 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. In addition, task A cannot distinguish this situation from a slowdown in task B caused by a temporary increase in activity on its processor.

Similar problems arise throughout Ada, both in explicit communication such as the rendezvous, in implicit communication such as task activation and termination, and even in referencing shared variables. A detailed examination of these situations is given in [11].

The second way that a task may be affected by processor failure is the loss of part of its 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 the objects defined by one of its surrounding lexical blocks on another processor. A task in Ada relies on several contexts; the context of the body, the creator, and the masters [1]. All of these contexts may be different, and 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 and lost data. The remaining software may be affected by the failure and the effects include the permanent suspension of tasks on remaining processors for a variety of reasons, and the loss of contexts of some tasks. These effects could be quite extensive. As presently defined, Ada provides no definition of what software will be affected or how it will be affected; Ada has no failure semantics.

Selecting a Response

The selection of a response is made by the application program. In order to avoid a centralized system we suggest that each processor should contain a unit that will select the response for the processes on that processor. Clearly the information on which the response is based must be consistent across processors. Usually, the response will depend on the processes lost by the failure, so that the unit which selects the response should be able to obtain that information.

In Ada this could be done in several ways. In the example described below, the unit which selects the response on each processor is a task, which waits at an accept until a failure occurs. After a failure, the system calls the corresponding entry and passes the information about the loss of processes as parameters to the entry call.

Effecting a Response

The purpose of effecting a response is to provide replacement 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. For any programming language, the failure semantics must *guarantee* the existence of this remaining software. It there are no failure semantics or the failure semantics do not require adequate software to survive a failure, then it may not be possible to effect any response. To be useful, the states specified by the failure semantics for processes that remain but are damaged must be such that they allow some form of recovery. At the very least, it should be possible to remove such processes.

Although Ada has no failure semantics, for the purposes of discussion, we will assume that it does, and examine the subsequent issues that arise in effecting a response. Some facilities of the language can be used to effect a response while others limit what can be done.

In general it will be necessary for the software effecting the response to be able to communicate with the existing software. Clearly the software effecting the response cannot call an entry in an application task and wait for the task to reach the corresponding accept; it might never reach it. The alternative is for the application task to check with the software that effects the response whenever it reaches a state that could be affected by

failure. A better solution would be for the software effecting the response to be able to raise an exception in the application task. Although this is not possible in Ada, the effect of communicating with a task so that it can modify its behavior because of a failure can be obtained, at some additional cost, by aborting the original task and starting a replacement task which incorporates the modifications. The basic problem in Ada is that there is no way of getting the "attention" of a task unless the task cooperates.

Once the problem of communicating with a process has been overcome it is still necessary to program the desired modifications. Often the communication paths used before failure will have to be redirected. 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 more 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 a replacement 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, a replacement task has to be of the *same* type as the primary task and this makes the provision of degraded service difficult.

Although Ada allows the software effecting the response to create and start replacement tasks, the scope rules of the language require lexical placement of the replacement software within the scope of the software effecting the response. In fact, since the software effecting the response must be able to see all of the tasks on a processor, it may be necessary to spread this software across several tasks.

Consistent Data

When a failure is detected, the reconfiguration software on each processor will assess the damage, and select and effect the response. Each of these activities will normally depend on data surviving the failure and this data must be identical on *all* machines. For example, usually the selection of a response will depend on which processors and processes remain, but in many applications the choice will also depend on other variables and their values will have to be known. The altitude of an aircraft, for example, might determine the actions to be taken when part of the avionics system is lost. If different processors have different values for the altitude after a failure, the responses they select may work at cross-purposes.

Ada provides no facility for maintaining consistent data across machines. Pragma **shared** does require that all copies of a shared variable be updated whenever any copy is updated, but it does not guarantee that if a machine failure occurs during the update process either all or none of the remaining copies will have been updated.

IV DISTRIBUTION AND FAILURE SEMANTICS FOR Ada

It is possible to overcome the difficulties discussed above. The first step is to define distribution and failure semantics for Ada. 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 of Ada as closely as possible, and to allow the Ada syntax to remain unchanged.

Distribution Semantics

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. There is no theoretical reason to limit distribution to tasks, and we agree with Cornhill [7] that other objects could be considered. However, tasks are natural candidates for distribution and were intended for distribution in the requirements for Ada, and, for simplicity, we limit our discussion to tasks.

Failure Semantics

In section III 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 failure semantics be defined so that the state of the program following processor failure be identical to the state that would have occurred 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 failure semantics leads to a new problem; the status of the main program following failure. By definition all tasks in an Ada program whose masters are not library packages depend on the main program. 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 all except library tasks) will have to be aborted. This could effectively remove 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.

V A SUPPORT SYSTEM FOR FAULT-TOLERANCE IN Ada

Although Ada does not support the facilities required for continuation explicitly, fault tolerance can be achieved if the execution-time support structure is suitably modified to detect failure, implement the failure semantics described in section IV, and provide

consistent data. In this section we discuss the necessary modifications and in section VI 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 active, relying on regular liveness checks, or passive, relying on some form of timeout. We 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*.

Implementing Failure Semantics

The mechanism that we propose for implementation of the Ada failure semantics defined in section IV together with the heartbeat mechanism, is shown in figure 1. The failure semantics require that the loss of a processor have the same effect as aborting all tasks on that processor. Implementing this requires that the affected tasks be known and that the semantics of abort be applied.

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 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.

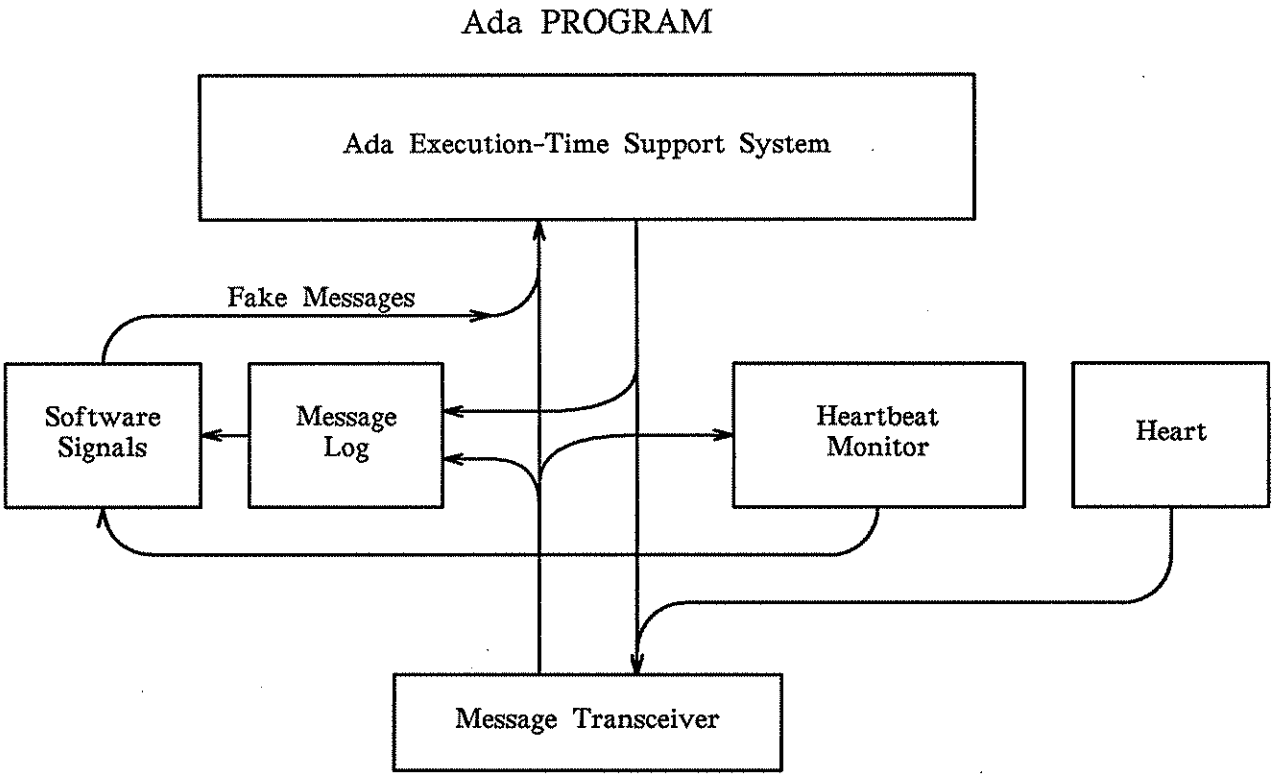

Fig. 1. Implementation Model For Each Processor

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, for example raising TASKING_ERROR, indicating that the serving task has been aborted and that rendezvous is not possible.

Provision of Consistent Data

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. A *two-phase-commit* protocol [12, 13] 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.

VI 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 III, 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.

Distribution

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 identification 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 loader directives for the designated task that would ensure that it is placed in the required machine. Alternatively, for tasks created by allocators, the distribution pragma could be required to appear in a declarative part; 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.

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.

The approach to signaling failure that we have chosen 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. This raises the problem of where the entry should be defined to ensure execution. We propose, therefore, that a dedicated task of the highest priority (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 the reconfiguration task is shown in figure 2. 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 (which must be guaranteed by the heartbeat mechanism to be the *same* 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.

We note that an exception cannot be used to signal failure. The difficulty is that Ada provides no way to ensure that the task in which the handler is defined will be executing

```
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. 2. Body Of Task RECONFIGURE_I.

when the exception is raised. What is required is a task that is idle until the failure has to be signaled. This is best achieved by having the task wait at an entry and signaling failure with an entry call.

Damage Assessment

As a consequence of the failure semantics defined in section IV, damage will be limited to lost tasks and the data associated with those tasks. No remaining tasks will be suspended, and all remaining tasks will have their complete contexts. However, the tasks and data that were lost need to be determined. Since the programmer controls distribution this is quite simple.

The information about which tasks are on which processors could be maintained and referenced in three ways: as a table maintained by the execution-support system; as a table maintained by the program itself; or maintained implicitly within the program. In the third case, 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.

Selecting and Effecting The Response

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 replacement 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 3.

Redirection of communication to replacement software that has been started following a failure has to be programmed ahead of time into tasks that call entries in tasks that might

```
task REPLACEMENT_SERVICE is
   pragma distribute(PROCESSOR_I);
end REPLACEMENT_SERVICE;

task body REPLACEMENT_SERVICE is
begin
   -- Code necessary to initialize this replacement 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
      -- replacement service. Any data required can be
      -- obtained from DATA_CONTROL_I.
end REPLACEMENT_SERVICE;
```

Fig. 3. General Form Of A Replacement Task.

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.

Consistent Data

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 recompute any expendable values. A sensor value, for example, is usually useful only 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 together would maintain a consistent set of essential data across all processors using a two-phase-commit protocol. When a task had produced some essential data, it would pass that data to the local data management task by calling an 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 completed satisfactorily. When essential data was required (typically for the initialization of a replacement 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.

Summary

In summary, an Ada program that uses the support system described in section V 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.

```
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.

(2) 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 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.

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

(4) One or more tasks on each processor designed to cope with reconfiguration on that processor; these tasks contain an entry with a parameter that indicates which hardware component has failed. These entries would be called automatically by the support system following failure detection.

(5) 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 on the processor to distribute essential data at any time.

VII IMPLEMENTATION

An important question that arises is the execution-time overhead that this scheme introduces. This overhead is determined by the particular data structure chosen for the log in any implementation, and the message volume generated by any particular application. We will use an example data structure in this section for the purposes of discussion. We do not claim that it is in any sense optimal.

The purpose of the message log is to implement failure semantics for Ada. However, other data structures will exist to implement distribution semantics and other execution-time facilities. In particular, a table describing the mapping of tasks to machines will be required. We will refer to this as the task/machine map. This map need not be stored in its entirety on each machine. Only that part needed by the tasks on a particular machine need be stored on that machine.

A simple implementation of the message log could consist of two two-dimensional arrays. Each array is indexed in one dimension by the various machines in the system, and by the local tasks of the machine that the log resides on in the other. One of these arrays, OUT, will be used to record information on outgoing messages that need a reply, and the other, IN, to record information on incoming messages that request entry calls.

An element of OUT is a list. It describes messages that a single local task has sent to a particular remote machine and for which that task expects a reply. When a message is generated it will contain a *message descriptor*, part of which describes the purpose of message. For messages that require a reply, the descriptor will also contain an *identification number*. A reply to a message will contain this same identification number as part of its descriptor. The message log can use the identification number to associate a message with its reply. Thus the only information that needs to be stored in the message list constituting an element of OUT is the descriptor of each message. The message list contains only the descriptors of messages sent to a particular remote machine for which a particular local task is awaiting replies. A given implementation may make it possible for a task to wait for two or more replies. However the maximum number of replies that it is possible to wait for would normally be very small. Apart from this implementation-induced waiting, the only occasion in Ada where a task can wait for more than a single reply is when one task is creating others and awaiting their elaborations.

An element of IN is also a list. It describes the entry calls received for a single local task from tasks on a particular remote machine. The list need only contain the

identification of the caller. A consequence of the rules of Ada is that any task can be on only one entry queue at a time thus the caller's identity identifies the call uniquely [1] The maximum length of a list in IN is the number of tasks in the program, and the average length normally will be much less.

There are three major operations that must be performed on the message log:

(1) Processing a message as it is transmitted.

(2) Processing a message as it is received.

(3) Processing the log when a failure is detected to generate the fake messages.

We will discuss each of these in turn.

Outgoing Messages

The destination of an outgoing message must first be checked against the task/machine map to see if the message is being sent to a machine that has already failed. If so, the descriptor of the message will be used to determine the appropriate fake message to send to the local task.

If the destination machine has not failed then the message descriptor is checked to see whether the message requires a reply. If it does an identification number is added to the descriptor and the descriptor is entered into OUT.

If the message is a reply to an entry call then the corresponding list element in IN has to be removed. The Ada rule stating that entries must be processed in order of arrival ensures that the element to be removed will be the first on the list so that it can be found immediately.

Incoming Messages

If an incoming message requests an entry call then the identification of the sender must be stored in IN. If a message is a reply then the list entry describing the original message is removed from OUT. The list of descriptors constituting the element of OUT

will have to be scanned.

On Failure

When a machine fails the machine/task table must be updated. The row of IN corresponding to the failed machine must be used to remove the corresponding elements in the appropriate entry queues. The row of OUT corresponding to the failed machine must be used to determine which fakes message to send. Finally, the fake messages must be sent to the appropriate local tasks.

Consistent Data

The overhead associated with consistent data depends strongly on the application. The only data that must be kept consistent for all applications, is the machine/task table. This table will need to be updated when a machine fails and when tasks are created, are aborted or terminate.

Normally, an application will need to keep consistent data available across machines so that continued service can be provided. The run-time cost depends both on the amount of data that must be kept consistent and the frequency of updates. Both of these factors can be controlled by the programmer. We expect that for many applications the most recent version of much of the data will not be needed for reconfiguration thereby reducing the potential overhead.

Heartbeats

The run-time overhead associated with heartbeats should be small. If broadcast messages are possible then n messages per interval would be needed. If broadcast messages are not possible, then n(n-1) messages per interval would be needed. Messages could be piggybacked onto existing messages requiring extra messages only when the normal message traffic between machines was not frequent enough. In any case n-1 incoming heartbeats would need to be processed by each machine per interval, however the processing of a heartbeat could be very fast as long as there is no machine failure.

Execution-Time Performance

The overhead associated with processing an incoming or outgoing message involves manipulating the local message log. This overhead is very small compared to the cost of actually sending a message. It will be dependent on the size of the message log. This size will be bounded by:

(max no. local tasks) * (no. machines)
* (max no. of replies a single task can wait for + max no. tasks)

and will usually be much less than this upper bound.

Upon machine failure, fake messages must be sent to local tasks requiring them. No message passing between machines is involved and the situation will occur only rarely. The overhead incurred is not important provided it is bounded and can be achieved within any real-time constraints imposed by a particular application.

The overhead discussed above does not involve any message passing between machines. However, in order to maintain consistent data across all the machines, it is necessary to send many messages. Thus there will be considerable overhead associated with maintaining consistent data. Recall that it is not just data from the application that must be kept consistent. The overhead associated with maintaining consistent data will add considerably to the cost of creating and terminating tasks since the message logs on all the machines must agree. This overhead cannot be avoided if the system is not to be centralized. The application-specific data that must be kept consistent will depend on the application. Again, the overhead will be considerable, but the amount of data and the frequency with which it is updated is under the programmer's control, and so can be kept to the minimum necessary for the application.

A Testbed

An implementation of the tasking, exception handling, and some of the sequential features of Ada incorporating the ideas described in this paper has been undertaken. The goal of the implementation is to demonstrate feasibility and to allow systematic, repeatable demonstrations of the recovery mechanisms. Thus it takes the form of a *testbed* that is heavily instrumented and allows extensive control of the Ada program under test. Simple

programs written with the structure described in section VI have been executed on the testbed and have survived arbitrary processor failures. It must be noted that, of necessity, programs running on the testbed do not execute rapidly.

The testbed provides an arbitrary number of abstract machines and interprets their instruction sets. Communication between the abstract machines is provided by a message passing mechanism implemented by the testbed. Each abstract machine contains a message log as described above and can support any number of Ada tasks. Extensive control and monitoring facilities are provided that allow, for example, Ada tasks to be started and stopped and abstract machines to be failed deliberately. Thus an Ada program can be brought to any achievable state by controlling the order of execution of the tasks, and any combination of abstract machines failed with the program in that state. The program's recovery can then be observed.

A translator translates Ada programs into abstract machine instructions. The testbed executes on a single VAX 11/780 or on a network of Apollo DN300 workstations. Further details of the testbed can be found in [14].

VIII CONCLUSION

Although modern fault-tolerant processors are extremely reliable, they may still fail through degradation or physical damage. In order to benefit from the flexibility of distributed processing, crucial systems must be able to deal with processor failures. In particular, when processor failures preclude the continuation of full service it should be possible to use the remaining processors to provide degraded service. As degraded service can only be specified by the programmer, the programmer must be concerned with both distribution and continuation. This puts certain requirements on any programming language used to program crucial applications for embedded systems.

We have defined two sets of such requirements; distribution semantics and failure semantics. Any language to be used for programming non-transparent continuation must have complete distribution semantics and complete failure semantics. Further, system support to detect failure and provide consistent data must be provided.

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.

Many of the difficulties arise because Ada has no distribution semantics and no failure semantics. We have shown, however, that it is possible to define adequate distribution and failure semantics for Ada, so that the language can be used to program fault-tolerant distributed systems. The new semantics require no additions or changes to Ada's syntax although, as we have described, an extensive support system is necessary to implement them.

The semantics we arrived at were chosen to fit an existing language, and, although they allow fault tolerance to be achieved, they are not ideal. It is clear that distribution semantics and failure semantics should be incorporated into a language at an early stage in its design.

REFERENCES

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

(2) 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.

(3) 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.

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

(5) 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.

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

(7) D. Cornhill, "Four Approaches To Partitioning Ada Programs For Execution On Distributed Targets", *Proceedings of the 1984 IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota, October 1984.

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

(10) A.J. Wellings, G.M. Tomlinson, D. Keeffe, and I.C. Wand, "Communication Between Ada Programs", *Proceedings of the 1984 IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota, October 1984.

(11) 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.

(12) 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.

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

(14) J.C. Knight and S.T. Gregory, "A Testbed for Evaluating Fault-Tolerant Distributed Systems", Digest of Papers FTCS-14: *Fourteenth Annual Symposium on Fault-Tolerant Computing*, June 1984, Orlando, FL.