An Ada[1] Implementation of the Network Manager for the Advanced Information Processing System

Gail A. Nagle
Technical Staff
The Charles Stark Draper Laboratory
555 Technology Square
Cambridge, Massachusetts 02139
(617) 258-2238

Introduction

The Advanced Information Processing System (AIPS) is a data processing architecture designed to meet the reliability requirements of space vehicle applications. The Charles Stark Draper Laboratory is presently building an AIPS proof-of-concept prototype[2]. Ada was selected as the programming language in which major system services would be implemented. One part of the AIPS architecture is a fault tolerant input/output network which is under the control of a software module called the Network Manager. Ada provides a user with a significant number of options for implementing a given aspect of a design. During the development of the prototype Network Manager, some language constructs were found to be particularly well suited for certain types of situations. In one case the language did not provide a desired feature. Experience with Ada as a programming language for this application will be described here.

Background

Using Ada

Training in Ada was accomplished by a combination of viewing a subset of video taped tutorials presented by Jean Ichbiah, Robert Firth, and John Barnes, participation in an in-house course in Ada using the Grady Booch text Software Engineering in Ada, and a lot of "learning by doing". Initially the work in the in-house course and the "learning by doing" were somewhat impeded by the absence of a reliable in-house compiler which supported full Ada. This problem was greatly reduced by the timely arrival of the Digital Equipment Corporation's (DEC) Ada compiler and development system for the VAX[3].

The microprocessor used in the prototype system is the Motorola 68010. Since a compiler which handled full Ada was not available for this machine, it was decided that initial design and development of programs would be done on the VAX using the DEC Ada compiler. This strategy was based in large part on the portability of Ada code and the fact that Ada compilers which target the 68000 microprocessor were expected to be available well within the development time of the

---

1 Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

2 This work is supported by NASA under JSC contract NAS9-17560.

3 VAX is a registered trademark of the Digital Equipment Corporation.

prototype system. Thus progress in designing and programming the various modules could continue unimpeded by artificial constraints in the language.

The AIPS System

The AIPS architecture is highly modular. The needs of a specific application can be met by selecting components from a set of hardware building blocks and software system services.

One such building block is a fault and damage tolerant input/output network which allows a data processing element (typically a Fault Tolerant Processor or FTP) to communicate serially with I/O devices. The network consists of a number of full duplex links that are connected by circuit switched nodes to form a conventional multiplex bus. In steady state, the network configuration is static and the circuit switched nodes pass information without the delays associated with packet switched networks. Since not all pathways are enabled, the network has a set of spare links which allow it to be reconfigured in response to a failure. A network may serve only one processing element or it may be shared by several processing elements which contend for access to the network. In the case of a network dedicated to one processing element, a unique network configuration is possible. Such a network may be divided into subnetworks which allow an application to conduct simultaneous I/O operations with redundant, parallel devices from each subnetwork. Network organization and operation is completely transparent to an application running on the system.

The system service which is responsible for the reliable operation of an I/O network is the I/O Network Manager. The Network Manager can be run in any processing element connected to the physical network to be managed. It performs network initialization, fault detection and isolation, reconfiguration to a fault free state, testing for latent faults and status reporting.

High level design objectives of the network manager software for the prototype include transparency to network users, adaptability to dynamically changing system configurations, portability within the system, and modularity. Ada language constructs have been found which support these design goals. A full Ada version of the design has been compiled and run on a VAX 8600 using DEC's Ada compiler. To facilitate testing on the VAX, an Ada simulation of the network has also been developed. Installation of the full Ada version on the AIPS Fault Tolerant Processor must await the release of a compiler which targets the Motorola 68000. However, a modified version of the network manager has been compiled on the VAX using the Telesoft 1.5 cross compiler and is awaiting system test and integration.

Implementing the Network Manager in Ada

Overview

The number of Network Managers which a system needs depends on the number of physical networks in use. This number can vary from system to system and within a system over time. However, the number of networks which can be managed from a given processing site is bounded by the number of physical I/O interfaces it has. For the prototype system this upper limit is six. Furthermore, when a network is partitioned into subnetworks, each partition requires its own I/O interface. Thus a given processing element could manage at most six networks and/or subnetworks. From the point of view of the Network Manager, there was no functional distinction between the control of a network and the control of a partition .

The Network Manager is a system service which would be provided on demand of the System Manager. The System Manager is another software module which coordinates all other

System Services. The software for an active Network Manager process would consist of two major parts: a data store describing the topology of the network to be managed and the coded algorithms to provide the functions described above. Specific information about the network topology (e.g. the number of nodes and links in the network) would not be available until run time. Thus two factors motivating the design were the need to be able to start and stop the process on demand, and the ability to manage a network topology which is to be determined at run time.

The fact that several networks could be managed in parallel from a given processor required a non-reentrant module to coordinate the starting and stopping of the various manager processes. However, each manager was itself an atomic unit, requiring only information about the topology to be managed for it to be off and running on its own. Thus the Ada package was used to implement the system service of network management on a particular processor. The Ada task type was chosen to conduct the logic of managing a particular network. Other Ada packages were used to coordinate access to information about the various network topologies in the system and to encapsulate the data format required for communication with the prototype network nodes. Finally, the need for keeping the System Manager apprised of the status of network components was met by another task type which provided mutually exclusive read/write operations to a protected object containing current status information. The relationship among these various components is graphically depicted in Figure 1.

Major Features of the Implementation

package IO_NETWORK_MANAGER

This package provides the capability to manage the fault tolerant network defined by the AIPS architecture. A user, in this case the System Manager, can then start or stop management of any network in the system. The software for this module would need to be resident in each processing site which could in fact manage a network.

The visible interface to this package is composed of two procedure calls, *START* and *STOP*. The calling process first designates the definition (i.e. the topology) of the network to be managed through its interface to the data base package. It then calls the *START* procedure. When this call completes, network management is underway and network status is available. The call to *STOP* is also preceded by a call to the database to designate the network to be stopped. When the call to *STOP* completes, management of the indicated network is terminated and all resources allocated to that process are restored to the system. Thus network status is no longer available for that network.

task type NETWORK_MANAGER

A task object is created in the body of *IO_NETWORK_MANAGER* for each I/O network to be managed from a particular FTP. If a network is partitioned into a number of subnetworks, each subnetwork will be allocated its own manager task.

The concept of a partitioned network was devised to allow applications to conduct I/O operations with redundant, parallel devices resident in separate partitions. Within each subnetwork are a certain number of spare links which allow failures to be repaired intrapartition. While such a repair is taking place, communications on the other subnetworks can operate normally. To support this feature, management of the I/O networks is not conducted synchronously. Each partition is under the control of its own task object which performs its functions independently of the other subnetworks.

Since the number of possible networks which a given processor can manage is known in advance and is a relatively small number (currently six), a table of access types to these task objects is declared within the package body. The *START* and *STOP* procedures described above have access to this table. The task object has three entry calls. Not surprisingly they are

start, stop and *start_status*. During the *start* rendezvous, the task object makes a local copy of its network definition. During the *start_status* rendezvous, the network manager task initializes the protected status object. This rendezvous is also used to synchronize the two processes which can access this shared status object; i.e. the status reader will not be able to read until the status writer has written at least once. The task proceeds to "grow" a network. It then enters a loop whereby it will either accept an entry call to *stop* or will periodically monitor the network for faults. If faults are detected during monitoring, fault isolation and reconfiguration logic is activated. An alternate approach to the monitor-maintain cycle currently under consideration would provide this activity on demand when communication errors are detected in communications conducted on the network for application functions. The call to *stop* causes the process to exit its loop and come to its natural end at which time its resources are explicitly deallocated.

package IO_DATA_BASE, package NODE_MESSAGE_FORMATTING and other data structure considerations

The numbers of various network elements , i.e. nodes, links, I/O devices,etc., can vary from network to network, but within a given network topology, they are static. The first approach to the data abstraction process focussed on defining types to contain network topology information. The basic connecting unit of a network is a node. The AIPS prototype node has five ports. Each port may be connected to another node, a processor interface unit or an I/O device interface unit. Hence information about the element adjacent to a given port could be contained in a discriminated record where the information stored would depend on the type of that element. Five such records grouped as an array could make up one field of a larger record containing other information about the given node. Finally, a collection of these node records would define a topology for a given network. This collection was also housed in a discriminated record where the discriminant was the number of nodes (which was given a default value) and the other field was an array containing that number of node records. This structure has the additional feature that objects of this type could be declared within the network manager task type and would upon allocation of the task object have the default value number of nodes. Later this object could be updated to reflect the actual number of nodes in the network to be managed. A major strength of this approach was that of run time reliability. The compiler generated checks will ensure the correct usage of this structure, i.e. the user cannot access a portion of the structure where values are meaningless. A simple array that is large enough to hold data for any case could be misused in this way. However, the major drawback to this design was that each object so declared was allocated enough memory to hold as many members as the maximal value of the type of the discriminant.

A second design solved this problem of wasted memory space while retaining the ability to dynamically create array objects with the correct number of cells. This design used an access type to an unconstrained array type. A variable of this type is declared in the body of the task type. The number of nodes and a pointer to an array of node records are passed as rendezvous parameters to the activated task. During the rendezvous, the object accessed by the local pointer is allocated with as many cells as there are nodes in the network. These cells are assigned values by applying the 'all' construct to the local access variable and the rendezvous parameter. The only feature that is lost with this solution is the ability to later change the number of cells in the object. Since this network topology is constant for the lifetime of the task, this feature is not necessary here.

The discriminated record array structure did prove useful in another application. Since the network is a shared resource, the various processing elements using the network must contend for access. To reduce the overhead of the contention processing, a set of messages are grouped together in what is called a "chain". Messages are sent to nodes in chains. However, the number of messages to be sent to the nodes will vary with the reason for the communication. Thus the number of messages in a given chain will vary. For example, when monitoring the network, all

the nodes are sent messages. When growing the network, only one or two nodes are sent messages. When reconfiguring the network or testing spare links, it may be necessary to send messages to several nodes in one chain so as not to leave the network in an inconsistent state for

other network users before completion of the reconfiguration or test. Thus objects containing node messages will vary in length during the life of the task. Rather than create an object for each possible length chain, an object of the discriminated record array type was used. In this situation,the cost in extra memory is relatively small since each node message is only six bytes long and the prototype network may contain at most thirty-two nodes; however, the extra flexibility facilitates processing.

An operation provided by the data base package allowed a significant reduction in the memory needed to store topology data as well as the need to ensure that multiple copies of data remain consistent. Any FTP connected to a network can manage that network. The definition of the network used by a manager is the same regardless of the processing site except for the particular nodes (called root nodes) which connect the site to the network. Given the array of node records described above and the identity of the FTP, it is possible to derive the root node information. Thus network definitions can be stored centrally without regard for local variations which are derivable on demand.

A final Ada feature which proved useful in the data abstraction process was the representation clause. The prototype node expects to receive a message containing six bytes of data. Each byte in turn contains one or two bit wide fields which the node decodes to obtain its control information. Rather than having to remember that bits zero and one of byte three control whether or not a node is permanently reconfigured or only reconfigured for the next transmission, the representation clause allowed a type called *CONFIGURATION_LIFETIME* to be given two values, *ONCE_ONLY* and *PERMANENT*, with specific base two representations. The representation clause further allowed the node message type to be assigned to a specific two bit wide field for the lifetime information. Other fields in this record were named and positioned in a similar fashion. The programmer need not be concerned with masking and shifting to set up a node message. Code using these messages could be written more quickly and more reliably. Furthermore, the code becomes self-documenting and therefore easier to test. When the message needs to be stored in a general area of memory, unchecked conversion would allow the safe transfer of the byte organized information. This is the case when the message is written to a dual ported memory just prior to transmission on the network. Finally, this node dependent information was packaged as a unit which would shield the rest of the software from any necessary design changes in node hardware or protocol.

A Perceived Shortcoming in the Language

Ada currently does not allow a function to accept 'in out' parameters. While this makes sense in the context of a mathematical function, in the context of a computer program, a broader definition of 'function' can be supported. In this context, a function is a language construct that does something and returns a value as part of its call. In the network manager such a language feature would have been a great asset in conjunction with the short circuit 'and then' construct. During growth of a network, a node is subjected to a series of tests before it is formally added to the network. These tests are sequential in nature. If a node fails a test in the sequence, the remaining tests are doomed to fail and therefore need not be performed. A very elegant way of coding this testing sequence was:

```
if PASS_TEST_1
   and then PASS_TEST_2
   and then PASS_TEST_3
   and then PASS_TEST_4
 then ACTION;
 else OTHER_ACTION;
end if;
```

where the PASS_TEST_Ns are boolean functions. This code is easy to read and understand; it is also self-documenting.

The problem arose because each test needed to log error detection information as it was discovered. Since a global object was not desired here, other designs were examined. These included procedure calls for the tests within nested if then else statements and the calling of these procedures from functions declared locally within each subprogram performing the tests. However, none of these designs were so simple, straightforward or self-documenting as the original. It is hoped that this example will provide some additional motivation for a change in this restriction. Perhaps another type of subprogram would be the most acceptable solution.

Conclusions

From an implementation standpoint, the Ada language provided many features which facilitated the data and procedure abstraction process. The language supported a design which was dynamically flexible (despite strong typing), modular, and self-documenting. Adequate training of programmers requires access to an efficient compiler which supports full Ada. When the performance issues for real time processing are finally addressed by more stringent requirements for tasking features and the development of efficient run-time environments for embedded systems, the full power of the language will be realized.

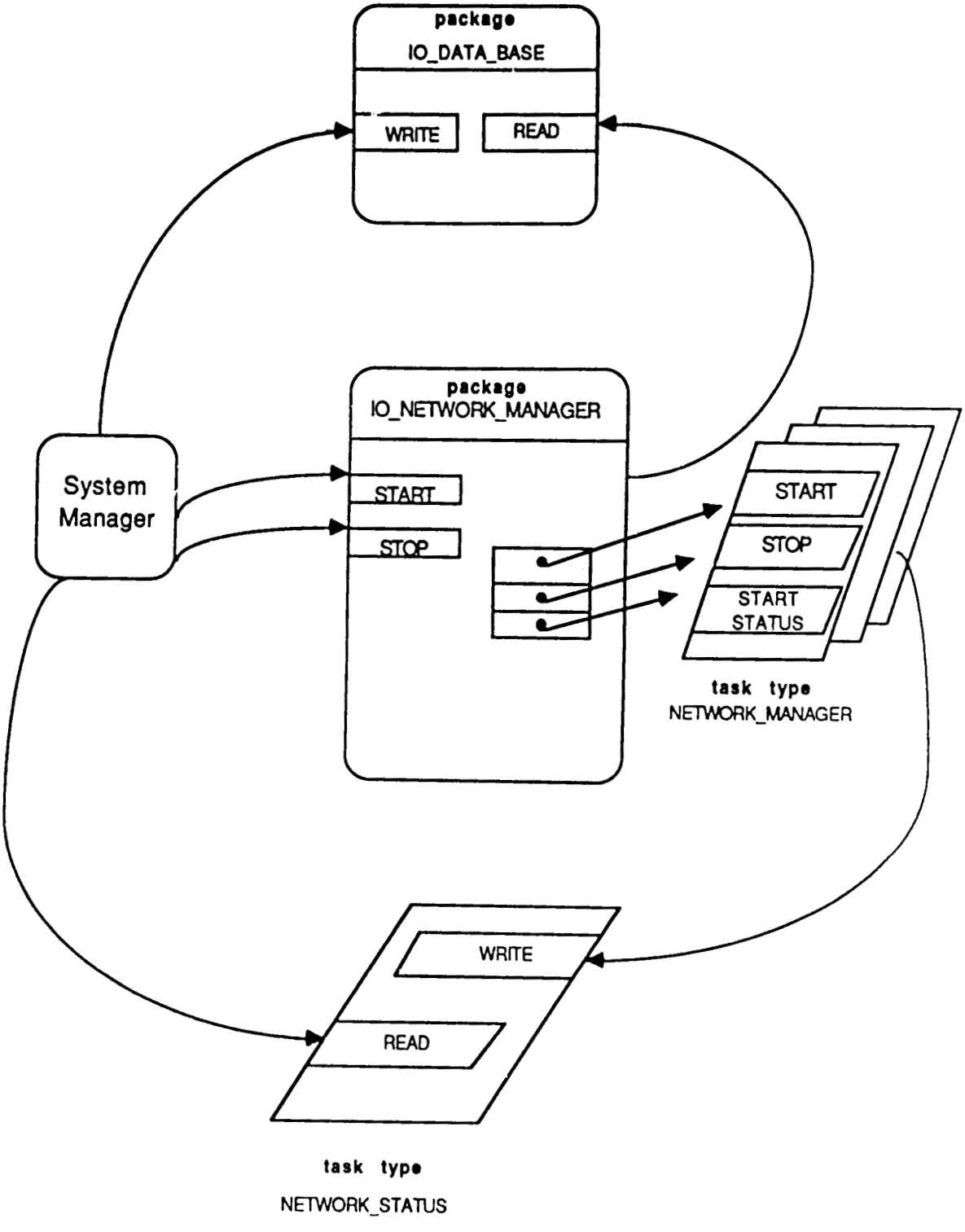

FIGURE 1: SOFTWARE COMPONENTS FOR MANAGING NETWORKS