ADAPT: Ada Distributed Application Prototyping Technique +

Joanne Z. Sattley*
Kirk Sattley*
Stuart C. Schaffner*
Stephen C. Schuman*
and
Edmund M. Clarke, Jr.

TR-20-81

+ Jointly issued as technical report by Massachusetts Computer Associates, Inc.

* Address: Massachusetts Computer Associates, Inc., 26 Princess Street, Wakefield, Massachusetts 01880

At Massachusetts Computer Associates, Inc., this research was supported in part by the U. S. Army CORADCOM, through the Scientific Services Program under Delivery Order No. 1704 from Battelle Columbus Laboratories.

At Harvard University, this research was supported in part by NSF Grant MCS-7908365 and by Contract N00039078-G-0020 with the Naval Electronics Systems Command.

Abstract

This paper discusses the development of an Ada Distributed Application Prototyping Technique (ADAPT) to support the rapid design and implementation of experimental distributed processing systems. This technique is predicated upon the use of Ada, with a set of systematically-generated library packages, to provide the facilities for interprocess communication.

ADAPT offers a prototyping environment in which a programmer can write and compile an Ada application that will run correctly on a distributed system. The specification of the allocation of tasks to individual processors in the system is decoupled from the specification of the application program itself; compilation of the program together with the specification of task allocation and various compile-time and run-time support packages produces object code for the separate processors.

The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official position, policy, or decision of the Department of the Army, the Department of the Navy or any agency of the U.S. Government, unless so designated by other documentation.

Table of Contents

1. Introduction

This paper discusses the development of an Ada Distributed Application Prototyping Technique (ADAPT) to support the rapid design and implementation of experimental distributed processing systems. This technique is predicated on the use of Ada, with a set of systematically-generated library packages, to provide the facilities for interprocess communication.

ADAPT offers a prototyping environment in which a programmer can write and compile an Ada application that will run correctly on a distributed system. The specification of the allocation of tasks to individual processors in the system is decoupled from the specification of the application program itself; compilation of the program together with the specification of task allocation and various compile-time and run-time support packages produces object code for the separate processors.

The overall approach plans for multi-processor configurations from the outset in order to support distributed applications at the earliest possible date. The first phase of the project develops the prototyping facilities for a multi-programmed uniprocessor target configuration. The second phase extends phase I to encompass various distributed multi-processor configurations. A third phase would yield an enhanced system based upon user feedback and experience with the phase II system.

Phases I and II comprise the subject-matter in this paper. We discuss the principal components of ADAPT, those which exist and others which still need to be built, and how they can be integrated to form a comprehensive prototyping facility.

The paper is organized as follows: Section 2 sets forth the basic assumptions regarding the nature of distributed applications that underlie our approach. In section 3 we describe a compile-time framework which permits the programming of a distributed application to be carried out in much the same way as if it were intended for a uniprocessor. In section 4 we describe the remote procedure call facility that is required by our target architecture. Section 5 discusses the specific architecture that we have chosen for our system and the ramifications of that choice. The program development tools that we use are described in section 6. The paper concludes with a project summary in section 7.

2. Assumptions Regarding the Nature of Distributed Applications

This section outlines our basic assumptions concerning the nature of the distributed application systems to be programmed in Ada. Abstractly, we wish to conceive of some given target configuration onto which a certain application is ultimately to be mapped as a network of communicating "Ada Virtual Machines" (AVMs). Every such configuration may therefore be characterized in the first instance by an undirected graph, as depicted for example in Figure 2-1:

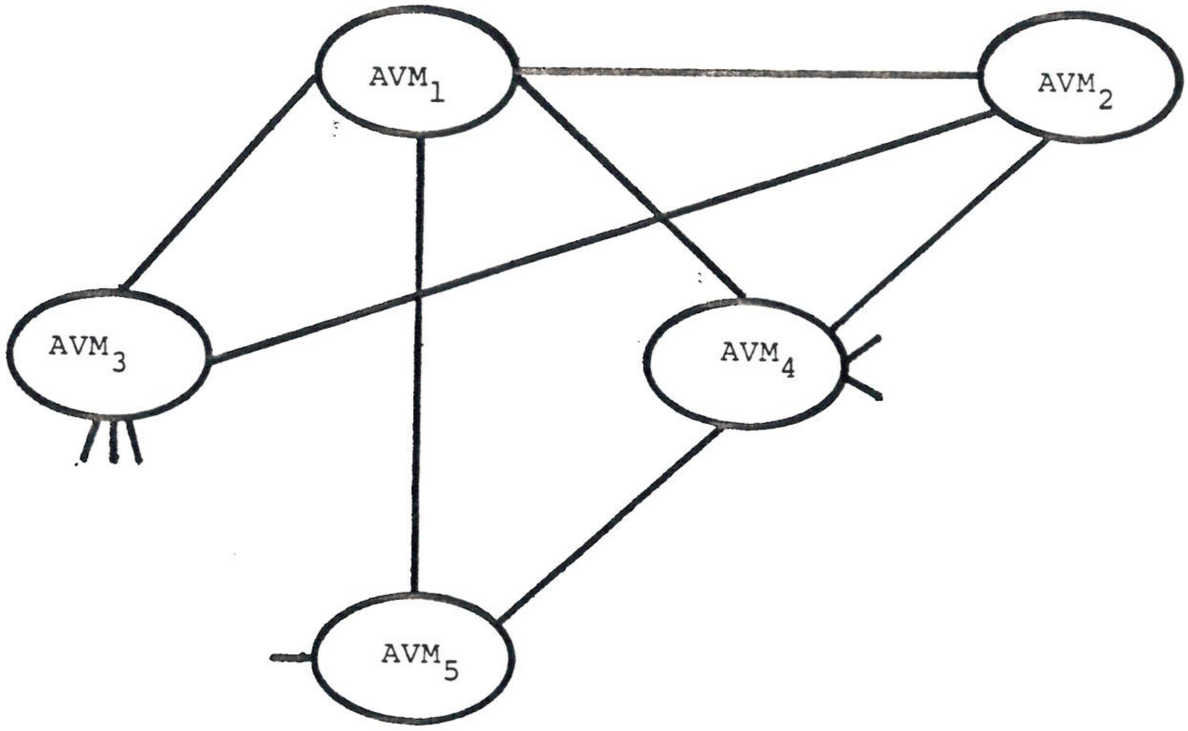

FIGURE 2-1: A network of communicating Ada Virtual Machines.

The individual nodes of a particular network correspond to fully independent (autonomous) processors, each of which is capable of executing a complete Ada program. Accordingly, an Ada Virtual Machine is to be viewed as an idealized single-processor environment that directly implements the run-time facilities required to support the semantics of the full Ada language. Thus the concept of an AVM embodies an abstract object machine for which Ada source programs might conventionally be compiled (but disregarding all dependencies upon a specific hardware architecture and/or host operating system); concretely, it may be thought of as providing its own address space, scheduler and real-time clock, together with a certain set of external interrupts, low-level device interfaces, etc. We refer to this environment as a "virtual" (rather than "actual") machine so as to also eliminate considerations arising from the fact that several such machines might sometimes be multiprogrammed on the same physical processor (e.g., in the context of an underlying time-sharing system).

The connecting edges appearing in a given network represent possible paths of bidirectional communication between distinct processor nodes. (Non-connecting edges, like those shown in Figure 2-1, are meant to suggest additional paths of communication, for instance with various devices attached to the individual virtual machines.) The connectivity of such a network is assumed to be sufficient for supporting the intended pattern of interprocessor communication, meaning that each edge corresponds to a path whereby both the requisite data and any appropriate control signals can by physically transmitted between the two connected nodes; moreover, the bandwidth of these connections is presumed to be adequate for the application at hand.

We shall assume that the target configuration for any specific application is always statically defined -- i.e., that the number of virtual (and even actual) processors is established once and for all, and that the necessary paths of communication exist from the outset. The primary stipulation which we impose is that all interactions between separate nodes of the network thereby defined must be achieved by explicit communication across these more or less "thin wire" connections. In other words, we preclude interactions based upon the existence of shared memory or any form of centralized control. This implies that the application in question must be formulated from the beginning as a distributed system. The issue we wish to address is how one might go about programming such applications in Ada, so as to be able to effectively map those programs onto the given multiprocessor configuration.

Ada provides an adequate basis for programming systems of communicating sequential processes, and for supporting synchronous communication between these processes. Once some desired pattern of logical communication has been established (for example, that depicted in Figure 2-2), there is no particular difficulty involved in formulating the specifications and subsequent definitions for the corresponding caller and server processes (or subsystems). Insofar as the resultant program is destined to be executed on a single processor configuration (as represented by the Ada Virtual Machine considered here), the job is effectively done once all of the separate compilation units comprised by that program have been successfully compiled (since an AVM is assumed to be capable of directly executing any complete Ada program, regardless of its textual decomposition).

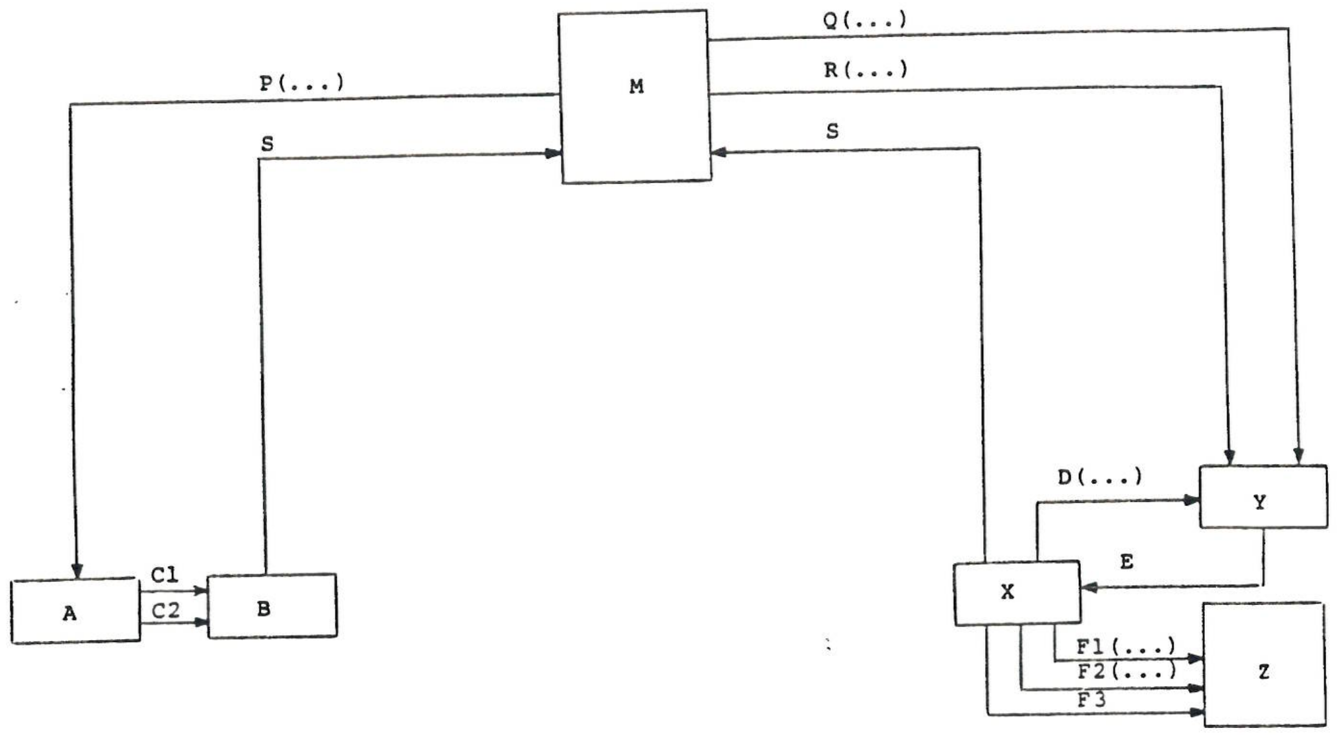

FIGURE 2-2: Example Application, in terms of Communicating Sequential Processes

However, when the target configuration is a network of interconnected AVMs (e.g., Figure 2-3), then it is far less obvious how to proceed. The effect that we should like to

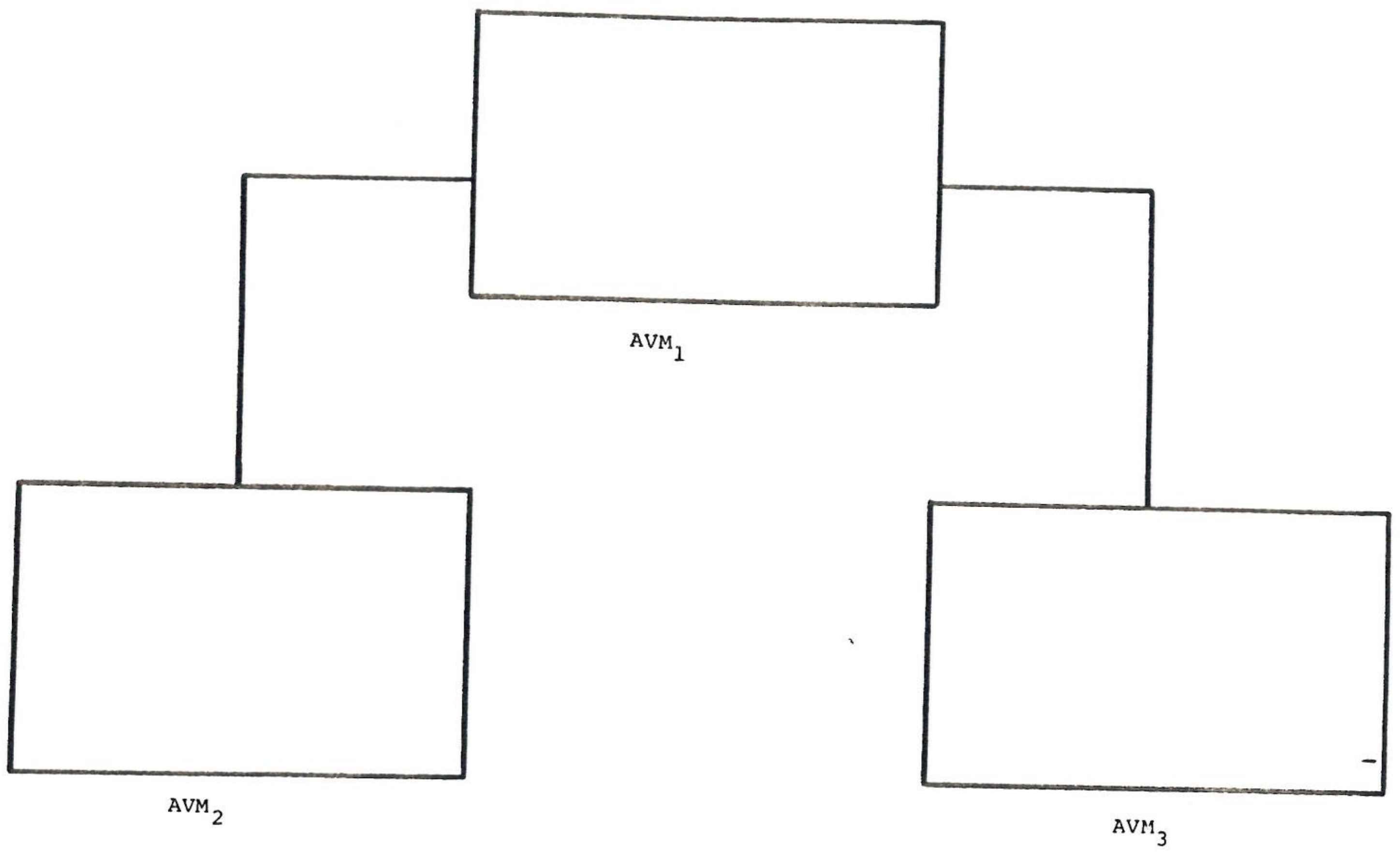

FIGURE 2-3: Example Target Configuration, in terms of interconnected Ada Virtual Machines

achieve is to be able to essentially "superimpose" the intended pattern of communication upon the underlying network (as suggested by Figure 2-4), thereby preserving the overall logical structure of the application. While the ability to do so presupposes that the application in question was formulated as a distributed system in the first place (i.e., based solely upon communicating sequential processes), it should then be possible to map that structure onto any appropriate target configuration, whether centralized or distributed. This is the premise of the approach outlined in the present paper.

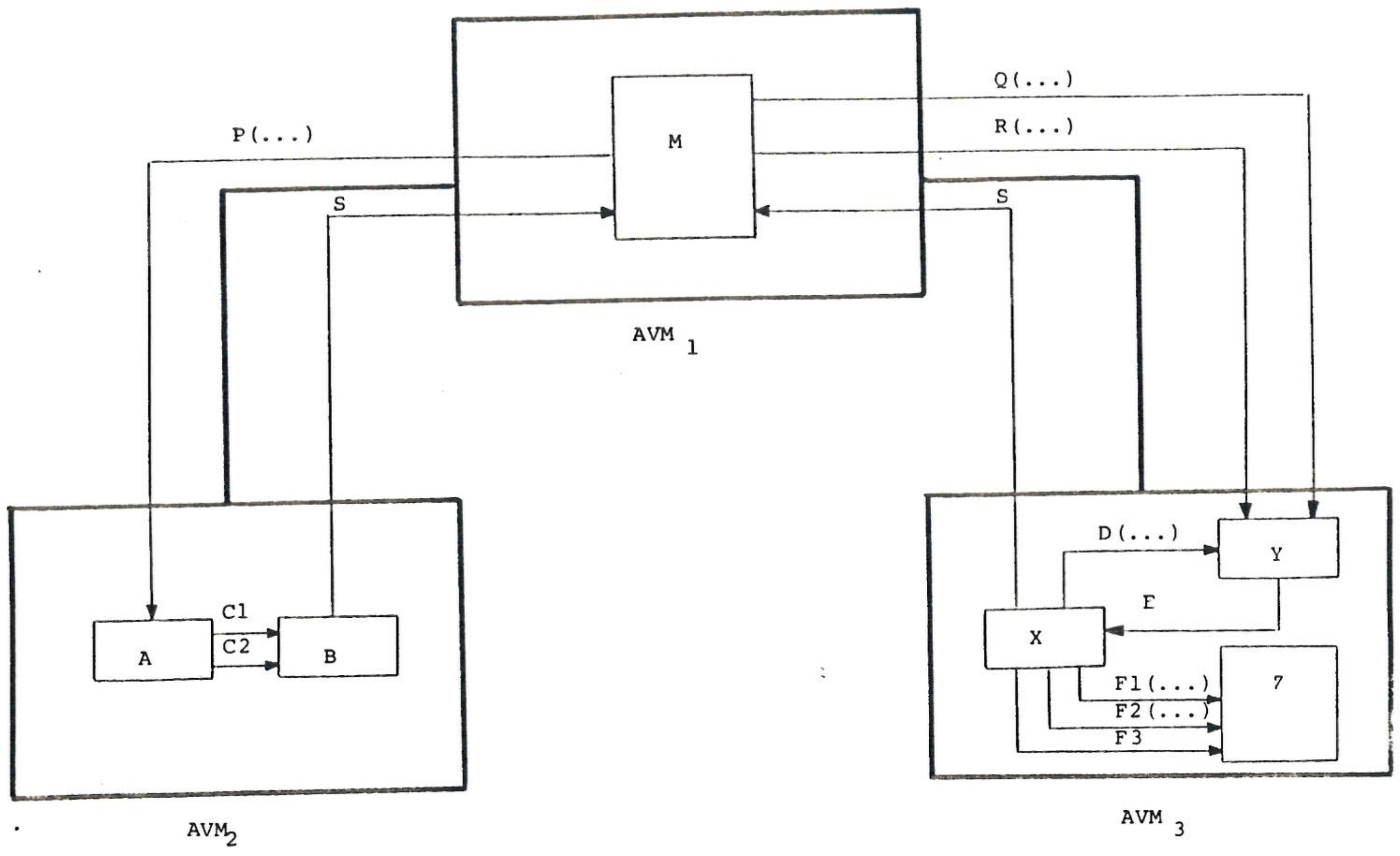

FIGURE 2-4: Superposition of Example Application upon the given Target Configuration

3. Ada Language Framework

In this section, we shall outline a basic approach to constructing a distributed application, such as that depicted in Figure 2-4, by making extensive use of the separate compilation facilities in Ada (and also of the related capabilities for generic program units). For expository reasons some practical problems associated with building distributed software will have to be glossed over (or neglected entirely). In particular, we shall be concerned solely with constructing a definition for the steady-state operation of a given application, even though it is well known that the issues involved in startup and shutdown of a distributed system are far more difficult to address. Needless to say, ADAPT must and will address these issues.

The package declaration that follows shows, in skeleton form, an initial specification for the application as a whole:

```
package Config is

   type NODE is (NN$1, NN$2, ..., NN$n);      -- Node Names
   type NSET is array (NODE) of BOOLEAN;      -- Set of Nodes

   package Node$1 is ... end;
   ...

   package Node$h is

      type OPER is (OP$1, OP$2, ..., OP$k); -- Op Codes
      -- other type definitions ...

      Host: constant NODE := NN$h;
      Conn: constant NSET := (... => True, others => False);
      -- other constant declarations ...

      generic
         Site: in NODE;
      package Service is
         procedure P$1 (...);
         ...
         procedure P$k (...);
      end Service;

   end Node$h;

   ...
   package Node$n is ... end;

end Config;
```

In order to formulate such definitions, we have adopted the (purely lexical) convention of writing names with an embedded dollar sign, so as to be able to refer to unique identifiers as if they were elements of a set distinguished by means of subscripts. For instance, the declaration of the enumeration type NODE is meant to suggest a range of values NN_1, NN_2, ..., NN_n, whereas in practice the individual values would correspond to application-specific mnemonic names (e.g., NN_h might be written as the Ada identifier "FileServer"). Also, P$1, ..., P$k denote the particular procedural services which that individual node provides.

This first specification consists primarily of package specifications for the constituent nodes of the overall configuration. The logical interface of each separate node comprises, in addition to various type and constant declarations, the declaration for a generic package Service, which will ultimately be instantiated within the definition of other (caller) nodes.

The associated body for the package Config, shown below, serves to establish the overall conventions which are common to all nodes. As such, it is primarily concerned with defining the underlying communications interface, by which information will be physically interchanged between distinct (virtual) machines within the configuration. These conventions are embodied firstly in a series of data type definitions, including:

- XREC, corresponding to a "transaction record" that contains at least an indication of the respective source and destination nodes for each transmission, as well as an encodement of the particular "operation code" for that particular transmission;

- XMIT, corresponding to a complete transmission, as delivered to or received from a local communications interface, which includes both an XREC component and an associated buffer (whereby argument or result data may be forwarded).

Two different types of transmission are distinguished at the communication level, namely Transmit Call (XC) and Transmit Response (XR), and the corresponding subtypes of XMIT are also defined (CALL and RESP, respectively).

Finally, the actual communications interface is specified in the form of two distinct generic packages, ChnDriver and ChnServer. Each of these has a number of generic parameters, in particular, an operation Request and an operation Deliver which will be bound in the context of their subsequent instantiations in order to carry out the necessary acquisition and disposition of transmissions over the underlying medium.

This interface is assumed to take full responsibility for setting and using the Orig and Dest Fields of the transaction record part of such transmissions. The details of these interfaces will not be further specified here.

```
 with Medium;
 package body Config is

    function Card(N:in NSET) return INTEGER range

                           0..NODE'POS(NODE'LAST)+1...;

    subtype OPID is INTEGER range 0.....;  -- Max Op Code

    type XREC is record
       Orig, Dest: NODE;
       ...
       Code: OPID;
       ...
    end record;

    type BUFF is ... ;
    type XTYP is (XC, XR);

    type XMIT(T: XTYP) is record
       X: XREC;
       B: BUFF;
    end record;

    subtype CALL is XMIT(XC);
    subtype RESP is XMIT(XR);

    generic
       From, To: in NODE;
       with procedure Request(C: in out CALL);
       with procedure Deliver(R: in RESP);
    package ChnDriver;

    generic
       From: in NSET;
       To  : in NODE;
       with procedure Request(R: in out RESP);
       with procedure Deliver(C: in CALL);
    package ChnServer;

    package body ChnDriver is ... use Medium; ... end;
    package body ChnServer is ... use Medium; ... end;

    package body Node$1 is separate;
    ...
    package body Node$n is separate;

end Config;
```

We now introduce analogous definitions for each separate node of our distributed configuration (the outline for that representing the Node$h is shown below). In this instance, however, such a step no longer constitutes an "extra" level of abstraction; rather, it is essential -- for this is the first place in which we permit actual instantiations (of code or data), since we have only now reached a level that corresponds to some physical machine environment.

The definition of such a shell serves to establish what might be construed as an "Application Virtual Machine", in terms of which the constituent subsystems of the actual application (e.g., the modules A$1...A$m) may then be programmed without further regard to the distributed nature of the underlying target configuration. This definition serves to provide:

- An indication of the target environment for this particular node (pragma SYSTEM);
- The specification of the application modules to be hosted within this node (the package declarations for A$1...A$m);
- A mapping of the remotely callable services provided by this node onto the operations defined by those modules (e.g., renaming of P$i);
- Definition of both sides of the higher-level protocol required to support such remote calls, namely the driver side (the body of the generic package Service) and the server side (the body of the non-generic package Support);
- Finally, instantiations of the remote services needed to implement the application modules of this node (package Node$u, Node$v, etc.).

```
separate (Config)
package body Node$h is

   pragma SYSTEM(...);

   -- Specify local application modules:

   package A$1 is
      procedure Q$1(...);
      ...
      procedure Q$f(...);
   end A$1
   ...
   package A$m is
      procedure Q$1(...);
      ...
      procedure Q$g(...);
   end A$m;

   -- Local (re)definition of services:
   ...
   procedure P$i(...) renames A$a.Q$b;
   ...


   -- Support services called remotely:

   package Support;
   package body Support is  -- Server side of Protocol
      ...
   end Support;

   package body Service is  -- Driver side of Protocol
      ...
   end Service;


   -- Provide services needed locally:

   package Node$u is new Config.Node$u.Service(Site => Host);
   ...
   package Node$v is new Config.Node$v.Service(Site => Host);


   package body A$1 is separate;
   ...
   package body A$m is separate;

end Node$h;
```

Within the framework of this shell, the application modules would again be defined as separately compiled subunits:

```
separate (Config.Node$h)
package body A$1 is

   ... Node$u.P$i(...) ...

end A$1;

...

separate (Config.Node$h)
package body A$m is

   ... Node$v.P$j(...) ...

end A$m;
```

The approach outlined above effectively makes use of the Ada "Program Library" to establish the context in which individual components of a distributed application may be defined in terms of a purely procedural interface to services which are nonetheless hosted on different nodes of a distributed target configuration. The possible protocols by which such an "interprocessor procedure call" capability might be realized are the subject of Section 4 of this paper.

4. Remote Entry and Procedure Call Protocols

In this section, we shall be concerned with the protocols by which the desired interprocessor procedure call capability is implemented for a particular distributed application. Thus, at this point, we shall elaborate upon actual definitions for the driver side (which serves to map such calls onto the communications interface) and the server side (which acts to carry out such calls on behalf of any remote caller); these implementations correspond to the bodies of the packages Service and Support, respectively, which are defined within the body for the node wherein those remotely callable services are to be hosted.

For purposes of exposition, we shall consider only one instance of such a definition, that associated with the virtual machine Node$h (which makes available the operations P$1...P$k) and, moreover, we shall sketch out the detailed implementation for only one of the operations in question, identified throughout as P$i. This involves no loss of generality, since the structure for all other operations and nodes is essentially the same. Accordingly, the overall goal for the implementations that will be described here is to provide the capability suggested by Figure 4-1, namely to permit application processes such as A_1, A_2, B...C, residing on separate (virtual) machines, to invoke the operation P_i hosted by $Node_h$ (corresponding to yet another such virtual machine) as though by a simple (local) procedure call.

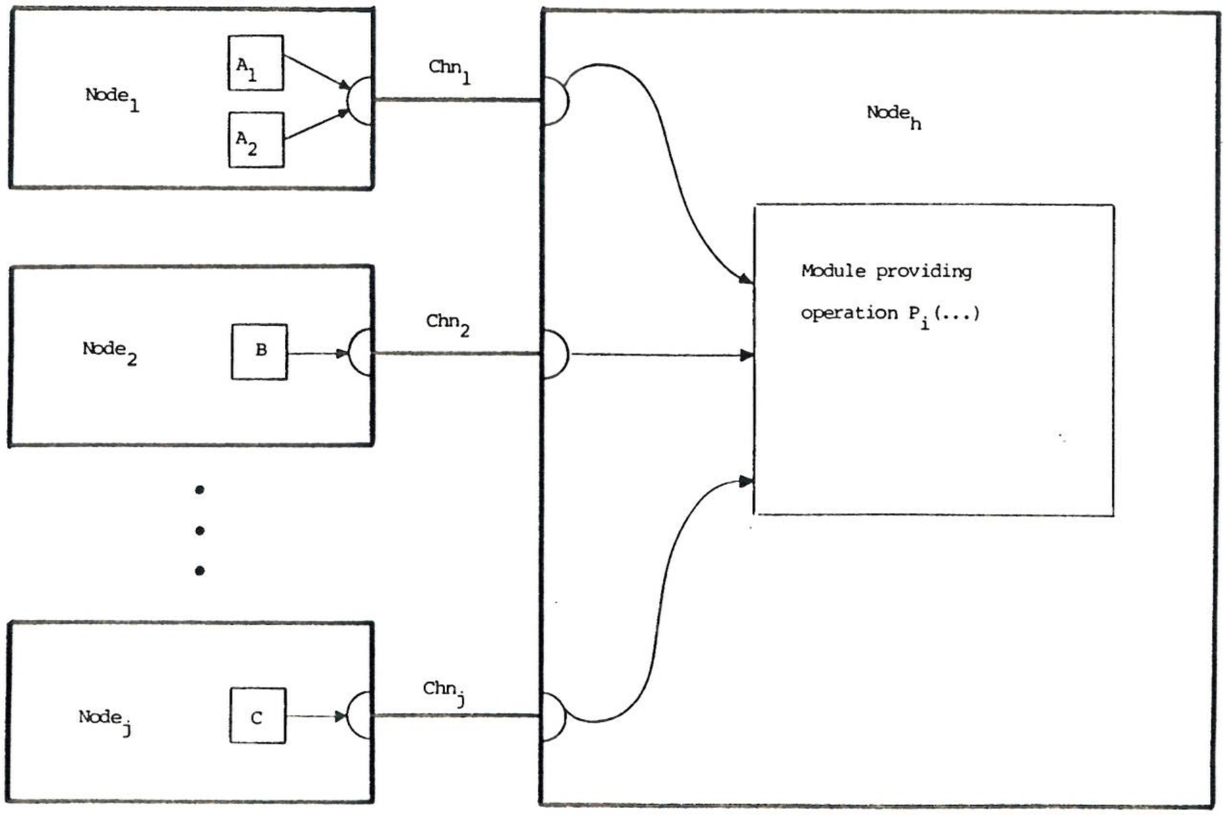

FIGURE 4-1: Overview of the Required Capability, to Support Remote Calls on the operation P_1

To simplify the presentation, we shall assume that the operation of interest has the following specification:

```
procedure p$i(A1: in TA1;...; Ax: in TAx; R1: out
          TR1;...; Ry: out TRy);
```

where Aj stands out for the jth input argument (of type TAj) and Rk stands for the kth output result (of type TRk); formal parameters of mode "in out" are thus presumed to have been decomposed into separate input and output objects. We note that some restrictions must be imposed upon the types of parameters in the present context. Specifically, it must be possible to copy the associated objects from one machine to another, which precludes the passage of task or "limited private" types (for which assignment is not defined). Similarly, it must be possible to meaningfully refer to such objects both locally and remotely, which precludes the passage of access types (except when declared as "private").

In the subsections which follow, we shall develop two alternative definitions for the desired protocol, referred to as the Remote Entry Call and the Remote Procedure Call, respectively.

In the first (and simpler) version, we impose the property that, from each distinct caller node, there is at most one remote call to any given operation in progress at a time. Such an implementation is appropriate, for example, in cases where the operations to be invoked are known to be entries (i.e., serviced in a purely sequential fashion), whence there is no advantage to be gained by forwarding more than one potentially concurrent call from some particular node (since these would then have either to be buffered within the communications medium or enqueued by the corresponding server node).

The second version relaxes this restriction, allowing a (bounded) number of calls on the same operation to proceed concurrently from within each separate caller node. This somewhat more complicated strategy is useful in situations where there is some optimization to be achieved (on the server side) by recognizing new calls before all previous ones have been completely serviced (as for instance in the context of a disk scheduler).

It must be stressed that there is no semantic distinction between these alternative implementation strategies. The choice affects only system throughput and thus the overall performance of the application in question; it should therefore be made on that basis alone.

We shall now proceed to develop Ada definitions for these two alternative protocols, expressed primarily in terms of the synchronous communication primitives embodied in the tasking facilities of that language. Each of the implementations to be described consists of the driver side (the body of the generic package Service, which is to be instantiated within one or more remote caller nodes), and the corresponding server side (the body of the package Support, which resides within the Ada Virtual Machine that hosts the operations in question).

We have further extended this approach so as to take into account the unreliability of the transmission medium in question, while still assuming that the nodes within the overall configuration are perfectly reliable. [See document CADD-8103-3102.]

4.1 The Remote Entry Call

As stated above, the first strategy is based on the property that no more than one remote call on each operation is in progress from the same node at any given time, so as to avoid saturation of the communications medium or overloading of the corresponding server node. As such, this property is necessarily established on the driver side of the protocol defined below.

4.1.1 The Driver Side

The overall structure and associated data-flow for the driver side are depicted in Figure 4-2. Calls on the operation P$i, originating from application tasks Ta...Tz are fielded by an Agent which is specific to that operation (AGTi); this latter acts to acquire the input arguments for each individual call (Al...Ax) and to subsequently deliver the corresponding output results (Rl...Ry). These two separate transactions for every operation hosted by Node$_h$ (P$1...P$k) are dispatched via distinct processes, the Driver Call Handler (DCH) and the Driver Response Handler (DRH), which respectively act to forward calls and retrieve responses from the Local Channel Driver (LCD) for Node$_h$. These handlers are formulated as independent (concurrent) processes so that the order in which LCD requests calls or delivers responses will not be unnecessarily constrained by this protocol.

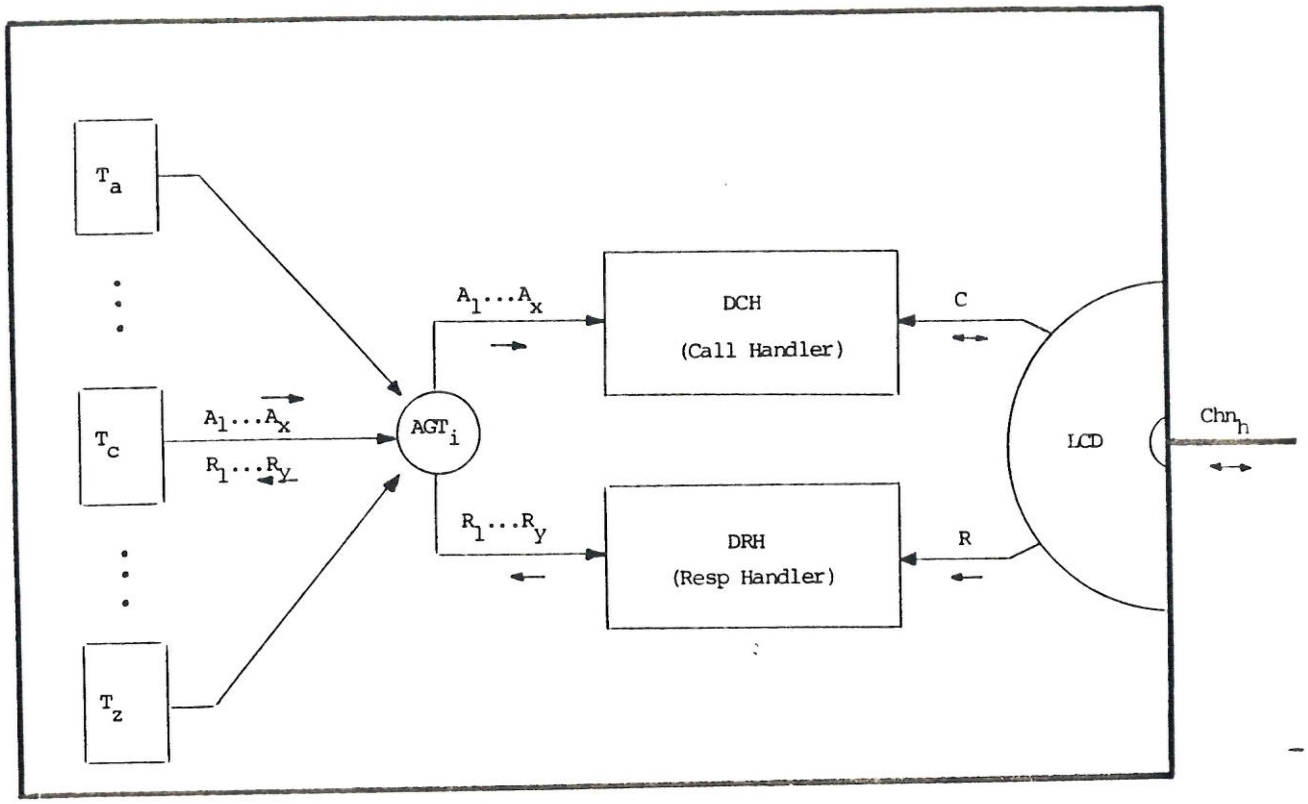

FIGURE 4-2: Overall Structure and Data-Flow on the Driver Side for the Remote Entry Call Protocol.

The outline of (generic) package body for the driver side is shown below:

```
package body Service is  -- Driver Side, defined in Config.Node$h:
   task DCH is
      entry ReqCall(C: in out CALL);
      entry DC$1(...);
      ...
      entry DC$i(A1: in TA1; ...; Ax: in TAx);
      ...
      entry DC$k(...);
   end;

   task DRH is
      entry DelResp(R: in RESP);
      entry RR$1(...);
      ...
      entry RR$i(R1: out TR1; ...; Ry: out TRy);
      ...
      entry RR$k(...);
   end;

   package LCD is new ChnDriver(
      From => Site, To => Host,
      Request => DCH.ReqCall,
      Deliver => DRH.DelResp);

   package D$1 is ... end;
   ...
   package D$i is
      procedure P(A1: in TA1;...;Ax: in TAx;
                  R1: out TR1;...;Ry: out TRy);
      procedure PutArg(B: in out BUFF;
                  A1: in TA1: ...; Ax: in TAx);
      procedure GetRes(B: in BUFF; R1: out TR1;
                  ...; Ry: out TRy);
   end D$i;
   ...
   package D$k is .. end;

   procedure P$1 (...) renames D$1.P;
   ...
   procedure P$k (...) renames D$k.P;

   ... + bodies of DCH, DRH, D$1, ..., D$k
end Service;
```

The handler processes DCH and DRH are directly specified in terms of Ada tasks, with entries to be called by the channel driver and by the agents for the remote operations to be invoked. LCD is obtained by instantiation of the generic definition associated with the overall configuration. For each operation, there is then a corresponding Driver package, D$1...D$k, which provides an operation P to be called by an application process (as P$i) along with operations for moving arguments into and results out of the actual transmission buffers.

Each time the channel driver requests a call (entry ReqCall), DCH makes a (non-deterministic) choice among the Agents waiting to deliver a call for one particular operation (entry DC$i), whereupon it sets the OpCode of the transaction record for that CALL and transfers the arguments into the associated data buffer.

Each time LCD delivers a response (entry DelResp), DRH decodes the Opcode appearing in the transaction record of that RESP and then accepts the pending response request from the agent for that operation (entry RR$i), transferring the corresponding result data.

4.1.2 <u>The Server Side</u>

The server side of the Remote Entry Call protocol is essentially symmetric to the driver side. The overall structure and associated data-flow for this side are shown in Figure 4-3. The Local Channel Server (LCS) forwards incoming calls from connected nodes to the Server Call Handler (SCH), and transmits the corresponding responses as dispatched by the Server Response Handler (SRH). As before, these handlers are formulated as independent processes (so as not to constrain the order of transactions with the underlying communications medium) and play a purely intermediary role. The actual calls to a locally supported operation P$i are performed by one of a number of Surrogate processes (SGTi), which act as stand-ins for the original calling processes within some other node. Thus, there exist multiple surrogates for each remotely callable operation, which serve both to "buffer" incoming calls and outgoing responses (along with their associated transaction records) as well as to invoke the actual operation in question (as provided by one of the application modules A1...Am supported by Node$_h$).

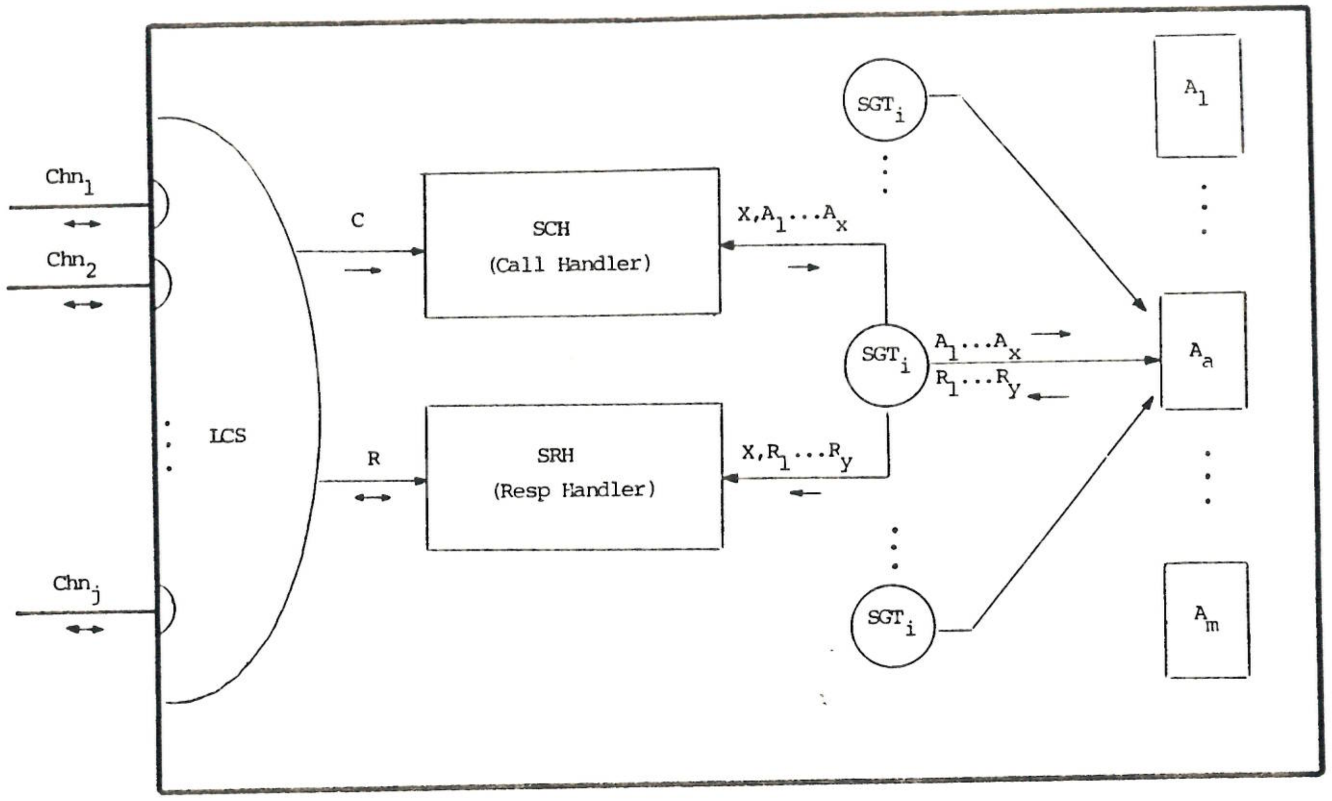

FIGURE 4-3: Overall Structure and Data-Flow on the Server Side for the Remote Entry Call Protocol

The implementation of the server side for Node$_h$ is defined in the (non-generic) package body Support, shown in outline form below:

```
package body Support is  -- Server Side, defined in Config.Node$h;

   task SCH is
      entry DelCall(C: in CALL);
      entry RC$1(...);
      ...
      entry RC$i(XR: out XREC; A1: out TA1;...; Ax: out TAx);
      ...
      entry RC$k(...);
   end;

   task SRH is
      entry ReqResp(R: in out RESP);
      entry DR$1(...);
      ...
      entry DR$i(XR: in XREC; R1: in TR1;...; Ry: in TRy);
      ...
      entry DR$k(...);
   end;


   package LCS is new ChnServer(
      From => Conn, To => Host,
      Deliver => SCH.DelCall,
      Request => SRH.ReqResp);

   package S$1 is ... end;
   ...
   package S$i is
      procedure GetArg(B: in BUFF; A1: out TA1;...; Ax: out TAx);
      procedure PutRes(B: in out BUFF; R1: in TR1;...; Ry: in TRy);
   end S$i;
   ...
   package S$k is ... end;

      ... + bodies of SCH, SRH, S$1, ..., S$k

end Support;
```

The handler processes are again directly specified as Ada tasks (SCH and SRH) and the communications interface is obtained by generic instantiation of the definition ChnServer for the overall configuration. As on the driver side, separate Server packages S$1...S$k are introduced here for each individual operation P$1...P$k that can be called remotely.

Upon delivery of a new call from LCS (entry DelCall), SCH switches on the OpCode and accepts a request for a call to the specified operation (entry RC$i) from the next of the (possibly many) Surrogates which are queued up on the corresponding entry. This dispatching consists simply of copying the transaction record contained within this particular CALL and transferring the associated arguments (via the operation PutArg provided by S$i).

Each time LCS requests a new response (entry ReqResp), SRH makes an arbitrary choice among pending responses ready to be delivered for any operation (entries DR$1...DR$k), whereupon the original transaction record and corresponding output results are copied into the RESP, to be transmitted back to the node from which that particular call originated.

It should be noted that no special precautions are taken on the server side to ensure the basic property of the Remote Entry Call protocol (at most one call in progress to each operation from any given node); this is solely a concern on the driver side. The servers simply invoke the local operations in question. If these have been specified as entries, then those calls will indeed be serviced sequentially; otherwise they will proceed concurrently.

What is of significance on the server side, however, is the fact that there are exactly as many Surrogates for each operation as there are Agents in total (distributed among the possible caller nodes). This property, referred to as local balancing, is fundamental to the solutions developed here, in that it ensures that this protocol does not require any additional storage capacity within the underlying communications medium nor any other form of buffering than that provided by the Surrogates themselves. This same property also guarantees that the communications interface will never by unduly tied up (since there will always be an available Surrogate ready to proceed).

4.2 The Remote Procedure Call

In this section, we develop an alternative to the Remote Entry Call protocol, wherein we allow a (bounded) number of calls to the same operation to be in progress concurrently within a given caller node (while still maintaining the overall load balancing that characterized our first solution). This somewhat more general strategy is described as a modification to the approach developed initially.

The point of departure for this strategy is to extend slightly the initial specification for the application as a whole:

```
package Config is

    type NODE is (NN$1, NN$2, ..., NN$n);
    type NSET is array (NODE) of BOOLEAN;
    subtype CONC is INTEGER range 0.....;     -- Max Concurrency

    package Node$1 is ... end;
    ...

    package Node$h is

        type OPER is (OP$1, OP$2, ..., OP$k);
        type MPLX is array (OPER) of CONC;
        -- other type definitions ...

        Host: constant NODE := NN$h;
        Conn: constant NSET := (... => True, others => False);
        Load: constant MPLX := ...;
        -- other constant declarations ...

        generic
           Site: in NODE;
           Usag: in MPLX;
        package Service is
           procedure P$1 (...);
           ...
           procedure P$k (...);
        end Service;

    end Node$h;

    ...
    package Node$n is ... end;

end Config;
```

The changes are wholly concerned with this added (potential) concurrency:

- A subtype CONC is introduced, whereby the maximum degree of concurrency anywhere within the system is specified;

- Within the package specifying each Node_h, a type MPLX is defined, values of which indicate a degree of concurrency on an operation-by-operation basis;

- A constant load (of type MPLX) is defined for each Node_h, whereby the limits on the overall concurrency (from all callers) are established for every such node;

- An additional generic parameter Usag (of type MPLX) is introduced for the Service package, so that the degree of concurrency for individual caller nodes may be set upon subsequent instantiation.

Minor modifications are also introduced into the body of the package Config, wherein the overall communications conventions are established:

```
with Medium;
package body Config is

   subtype OPID is INTEGER range 0.....;
   subtype RCID is CONC range 1..CONC'LAST;

   type XREC is record
      Orig, Dest: NODE;
      ...
      Code: OPID;
      Iden: RCID;
   end record;

   type BUFF is ... ;
   type XTYP is (XC, XR, AC, AR);
   type XMIT(T: XTYP) is record
      X: XREC;
      B: BUFF;
   end record;
   subtype CALL is XMIT(XC);
   subtype RESP is XMIT(XR);

   generic
      From, To: in NODE;
      with procedure Request(C: in out CALL);
      with procedure Deliver(R: in RESP);
   package ChnDriver;

   generic
      From: in NSET;
      To  : in NODE;
      with procedure Request(R: in out RESP);
      with procedure Deliver(C: in CALL);
   package ChnServer;

   package body ChnDriver is ... use Medium; ... end;
   package body ChnServer is ... use Medium; ... end;

   package body Node$1 is separate;
   ...
   package body Node$n is separate;
end Config;
```

The changes are to define an additional subtype RCID, which will serve to identify a particular remote call originating from a given node (since the OpCode alone will no longer be sufficient for this purpose), and to add a new component Iden (of type RCID) to all transaction records.

The only changes within the definitions of the separate nodes of the application would be to suitably set the generic parameter Usag upon each instantiation of the package Service:

```
separate (Config)
package body Node$h is

   pragma SYSTEM(...);

   -- Specify local application modules:

   package A$1 is
      procedure Q$1(...);
      ...
      procedure Q$f(...);
   end A$1
   ...
   package A$m is
      procedure Q$1(...);
      ...
      procedure Q$g(...);
   end A$m;

   -- Local (re)definition of services:
   ...
   procedure P$i(...) renames A$a.Q$b;
   ...


   -- Support services called remotely:

   package Support;
   package body Support is   -- Server side of Protocol
      ...
   end Support;

   package body Service is   -- Driver side of Protocol
      ...
   end Service;


   -- Provide services needed locally:

   package Node$u is new Config.Node$u.Service
           (Site => Host, Usag => ...);
   ...
   package Node$v is new Config.Node$v.Service
           (Site => Host, Usag => ...);


   package body A$1 is separate;
   ...
   package body A$m is separate;

end Node$h;
```

4.2.1 The Driver Side

The changes on the driver side in going from the Remote Entry Call to the Remote Procedure Call are concerned with keeping track of the identity of calls in progress. At the first level, this involves adding an additional ID parameter to the DC$i entries of the Driver Call Handler (DCH), and introducing a Post Response procedure (PR) to each of the Driver packages D$1...D$k:

```
package body Service is  -- Driver Side, defined in Config.Node$h:

    task DCH is
       entry ReqCall(C: in out CALL);
       entry DC$1(...);
       ...
       entry DC$i(ID: in RCID; A1: in TA1; ...; Ax: in TAx);
       ...
       entry DC$k(...);
    end;

    task DRH is
       entry DelResp(R: in RESP);
       entry RR$1(...);
       ...
       entry RR$i(R1: out TR1; ...; Ry: out TRy);
       ...
       entry RR$k(...);
    end;

    package LCD is new ChnDriver(
       From => Site, To => Host,
       Request => DCH.ReqCall,
       Deliver => DRH.DelResp);

    package D$1 is ... end;
    ...
    package D$i is
       procedure P(A1: in TA1;...;Ax: in TAx; R1: out TR1;...;Ry: out TRy);
       procedure PutArg(B: in out BUFF; A1: in TA1: ...; Ax: in TAx);
       procedure GetRes(B: in BUFF; R1: out TR1; ...; Ry: out TRy);
       procedure PR(ID: in RCID)
    end D$i;
    ...
    package D$k is .. end;

    procedure P$1 (...) renames D$1.P;
    ...
    procedure P$k (...) renames D$k.P;

    ... + bodies of DCH, DRH, D$1, ..., D$k

end Service;
```

The definition of DCH is then modified to store the identity of each call as part of the transaction record which it forwards. The corresponding modifications to DRH involve its passing that identity to the appropriate PR procedure prior to accepting a request to dispose of each incoming response:

Within a Driver package D$i, the modifications consist primarily of introducing a multiplicity of Agents for the same operation (whereas there was only one heretofore). This is accomplished by defining an array of agent tasks (AT), the range of which is established by the Usag generic parameters. Thus, the index in this array (of type AID) will serve to uniquely identify a particular call-in-progress for the operation P$i. At the same time, additional entries have to be provided for the AGT task: these are Init (whereby an Agent acquires its own identity) and Done (whereby it may be notified that the response for the call it is carrying out has been received). The procedure PR is essentially a call to this latter entry. A further task, the Agent Manager (AM) is now needed to establish the initial correspondence between the original call (from some application process) and the particular agent which will perform that transaction. This correspondence is created by the procedure P, which is called (concurrently) by every application process seeking to invoke the remote operation P$i.

After initialization an Agent enters its main cycle, wherein it first makes itself available to AM prior to accepting the resultant call via its entry Exec. Within the corresponding rendezvous, it delivers its own identity to SCH along with the arguments for the call in progress, it then awaits notification (via the entry Done) that the response for that particular call has been received before proceeding to request the results on behalf of the original caller.

4.2.2 The Server Side

In passing from the Remote Entry Call to the Remote Procedure Call protocol, essentially no modifications are required on the server side (since this latter already provided for some degree of concurrency, insofar as it had to handle incoming calls from more than one caller node). The only provision that must be made is to possibly increase the number of Surrogates for each operation P$i, which would be specified within the corresponding Server package S$i as follows:

```
subtype SID in CONC range 1..LOAD (OP$i);
```

thereby fixing the number of elements in the array of surrogate tasks. This will presumably preserve the overall load balancing (number of Surrogates = total number of Agents, for each operation Pi) upon which both of the protocols developed in this section have been based.

5. Hardware Support for ADAPT

The hardware which supports ADAPT must perform certain functions:

1. It must be able to execute Ada programs.

2. It must support distributed processing.

3. It must provide the user with interactive support for prototyping.

4. It must allow any prototype so developed to be tested, run and measured.

5. It must support interfaces between prototype systems and real systems.

The specific hardware configuration chosen makes use of commercially-available components wherever possible. The basic architecture is a heterogeneous collection of processors connected together in a network. This collection of processors will include at least one interactive personal workstation, a number of Intel 432 microprocessors, and access to a VAX/UNIX system. These processors will be connected by an Ethernet local area network. As needed, some of the 432 microprocessors can be interconnected by an Intel Multibus to allow high-bandwidth sharing of data.

This collection of hardware supports an Ada runtime environment, and each processor acts as an Ada virtual machine within that environment. The Ada runtime environment defines the low level network protocols required. Higher level network functions will make use of the Remote Entry and Procedure Call Protocols described earlier.

The personal workstation will support rapid and accurate interaction between the ADAPT system and the user. This interaction will involve both text and graphics, each where most appropriate. For example, individual Ada modules are textual in nature, but the configuration of individual modules into a complete system is inherently graphical.

The Intel 432 microprocessors will be the principal source of distributed processing power in the ADAPT system. They have been expressly designed to support Ada virtual machines within an Ada runtime environment. By connecting processor units in parallel, the computational power of a given network node can be varied by as much as a factor of 10. Intel claims that this can be done without the need for any software reconfiguration. This covers the performance range from a standard minicomputer to a small mainframe. An Intel 432 processor is available for the standard Intel MULTIBUS and is compatible with the Intellec software development system. Intel offers a prototype Ada compiler now which runs on a VAX and produces code for a 432. Intel plans to offer a full Ada for the 432 in the near future. Alternatively, the NYU Ada Translator could be fitted with a code generator for the 432. Like the current Intel offering, this would run on a VAX.

A complete ADAPT system will include a connection to a VAX, in order to take advantage of the many tools already available on it. It is still an issue how thoroughly the VAX can be integrated into the Ada runtime system which will control the 432 processors and still provide a compatible environment for tools which were built for UNIX or VMS.

The Ethernet was chosen for the Local Area Network because it has adequate performance and is (or soon will be) available from multiple sources. In particular, Intel will offer a MULTIBUS-compatible Ethernet board and DEC plans to offer Ethernet service for the VAX.

6. Program Development Tools

The principal program development tools offered by ADAPT are those which are supported as a matter of course by ordinary, centralized systems -- plus distributed-application-specific tools. Our long-range goal is to provide the programmer of distributed application programs with an integrated set of tools which are capable of manipulating the programs while they are under construction, in test phases, and in operation.

The figure below, Figure 6-1, depicts the interrelationship of the ADAPT components we have discussed in this paper. Let us point out here that ADAPT itself is a distributed application; conceivably, any of the components (the Execution Support Package , e.g.) might be broken into program pieces running on multiple processors as part of the Target Configuration.

The Target Configuration is based upon the concept of an Ada Virtual Machine (AVM) as defined in section 2 of this paper. The AVM provides us with an abstract Ada object machine completely programmed in Ada. Its specialized architecture supplies:

- an interpreter for high-level "A-code" (for computational processes), and
- operations for low-level support (tasking, i/o, run-time).

Within ADAPT, the target system configuration expands into a network of communicating AVMs which we show as a set of interconnected Target Processors (TP), each of which has associated Devices (DEV) and Application Workstations (AWS).

Using Gandalf as the environment shell, for example, furnishes an already well-developed user interface in addition to an integral program-development database (for housekeeping versions, revisions, etc.). It lacks, however, the requisite mechanisms for specifying target configurations, for downline loading of the target machines, multi-processor execution support, and so forth. These will need to be built.

Of course, a high-quality Ada compiler -- with appropriate code generators -- is an integral part of the prototype facility we are developing. Programming support tools will be needed to provide:

- the compilation framework (separate node definitions),

Overview of ADAPT Architecture

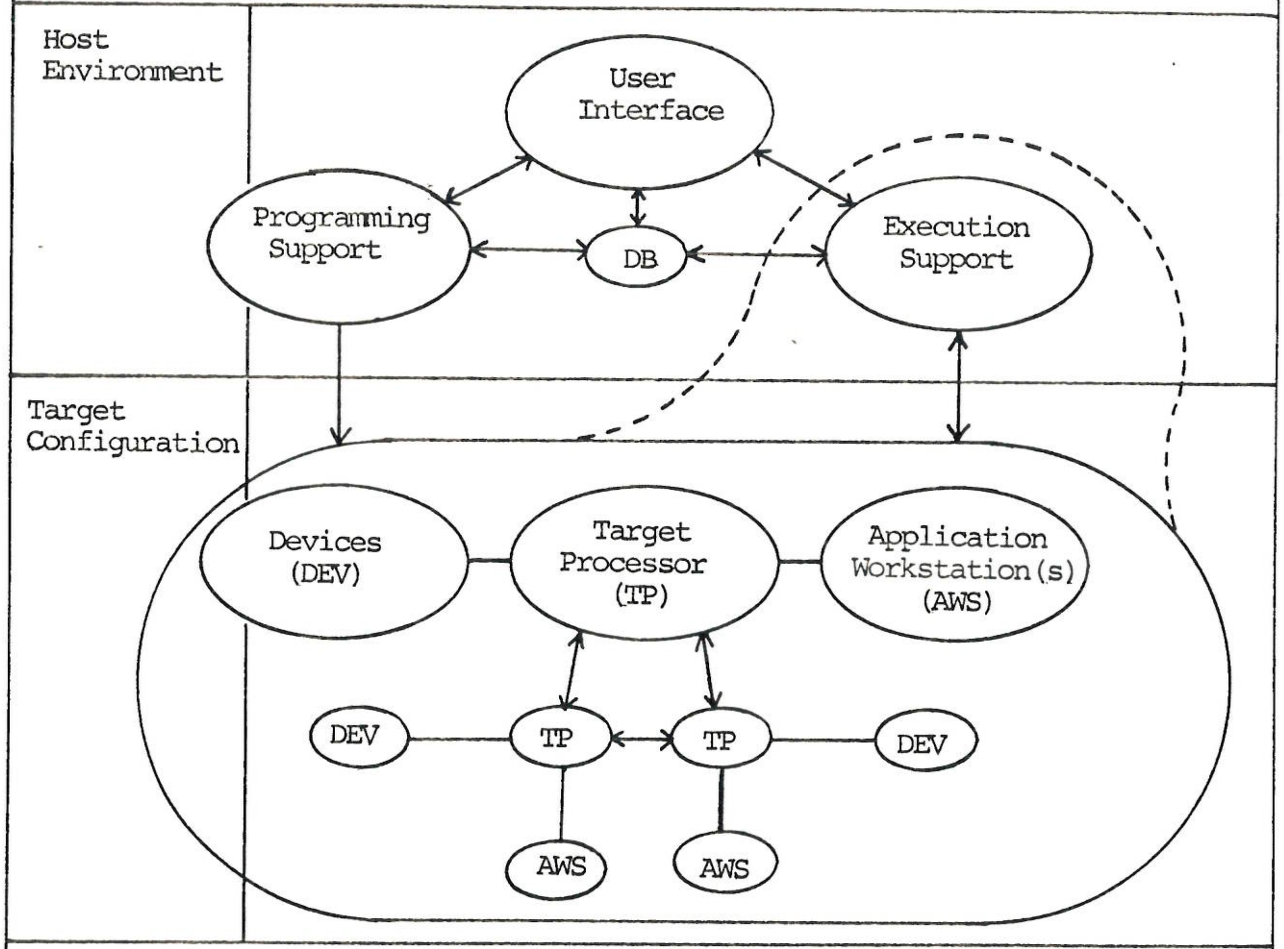

FIGURE 6-1

Bibliography

Buxton, J.N. and Druffel, L.E.; Requirements for an Ada Programming Support Environment: Rationale for Stoneman; Proceedings of COMPSAC; 28-30 October 1980.

Department of Defense Requirements for Ada Programming Support Environments "STONEMAN", U.S. Department of Defense, February 1980.

Fisher, G., New York University, Private communications regarding the Ada Translator.

Habermann, A.N., "An Overview of the Gandalf Project", in Carnegie-Mellon University Computer Science Research Review, 1979.

Hoare, C.A.R., Communicating Sequential Processes, CACM, August 1978, Volume 21, 8.

Ichbiah, J.D. et al., Rationale for the Design of the Ada Programming Language, SIGPLAN Notices, June 1979, Volume 14, 6, B.

Reference Manual for the Ada Programming Language Proposed Standard Document, U.S. Department of Defense, July 1980.

Schuman, S.A., Clarke, E.M., Nikolaou, C.N., Programming Distributed Applications in Ada: A First Approach, Massachusetts Computer Associates, Inc., CADD-8103-3102.

Schuman, S.A., Tutorial on Ada Tasking, Volume I: Basic Interprocess Communication, Massachusetts Computer Associates, Inc., CADD-8103-3101.

Schuman, S.A., Clarke, E.M., Nikolaou, C.N., Programming Distributed Applications in Ada: A First Approach, Tenth Annual International Conference on Parallel Programming, Bellaire, Michigan, August 25, 1981.

Ziegler, S., et al., Ada for the Intel 432 Microcomputer, Computer, Volume 14, No. 6, June 1981, pp. 47-56.