PROGRAMMING DISTRIBUTED APPLICATIONS IN ADA:
A FIRST APPROACH

by

Stephen A. Schuman
Massachusetts Computer Associates, Inc.
Wakefield, Mass. 01880

and

Edmund M. Clarke, Jr.
Christos N. Nikolaidou
Center for Research in Computing Technology
Harvard University

Abstract -- This paper addresses the problem of programming distributed systems within the framework of the Ada language, which provides primitives for interprocess communication based upon the model of Communicating Sequential Processes. We first discuss our basic assumptions concerning the underlying target configuration, the physical communication medium which is to support that application and pattern of the logical communication within the application proper. We then develop a first approach for constructing such applications using the separate compilation facilities of Ada. Finally, we consider two possible protocols for implementing the requisite distributed interprocess communication, referred to as the Remote Entry Call and the Remote Procedure Call, respectively.

1. Introduction

This paper addresses the problem of programming distributed applications within the framework of the Ada language [1,2,5]. Our ambitions here are confined to outlining a first approach in this area, whence a number of significant issues associated with the construction of such software arise, of necessity, deferred. We begin in Section 2 by setting forth the basic assumptions which underly the overall approach described herein. Section 3 is concerned with establishing an appropriate compile-time framework, within which the programming of an application destined for a multiprocessor target configuration can be carried out in such a way as one intended for a uni-processor target. In the final section, we turn to the development of protocols to support the requisite "interprocess procedure call" capability, so that the applications of interest can then be

\[\text{fig}

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 NCS-7908365 and by Contract N00039078-G-0020 with the Naval Electronics Systems Command.

2. Basic Assumptions

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 first instance by an undirected graph, as depicted for example in Fig. 2-1:

\[\text{fig}

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 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

0190-3918/81/0000/0038 $00.75 © 1981 IEEE
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 Fig. 2-1, are meant to suggest additional paths of communication, for instance with various devices attached to the individual virtual machines; however, interactions with purely local resources of this sort are of no direct interest here, and so will not be further discussed.) 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 be 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 [1], and for supporting synchronous communication between these processes. Once some desired pattern of logical communication has been established (for example, that depicted in Fig. 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).

However, when the target configuration is a network of interconnected AVMs (e.g., Fig. 2-3), then it is far less obvious how to proceed. The effect that we should like to achieve is to be able to essentially "superimpose" the intended pattern of communication upon the underlying network (as suggested by Fig. 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.
3. Overall Framework

In this section, we shall outline a basic approach to constructing a distributed application, such as that depicted in Fig. 2-4, by making extensive use of separate compilation facilities in Ada (also of the related capabilities for generic program units). The framework to be developed here must be regarded as simply a first approach to the problem, whereas many practical aspects associated with building distributed software will have to be glossed over (or neglected entirely) in the current context. (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.) This approach nonetheless provides a number of important insights into the nature of the problem itself.

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

```ada
package Config is
    type NODE is (NN段时间1, NN段时间2, ..., NN段时间n);  -- Node Names
    type NSET is array (NODE) of BOOLEAN;  -- Set of Nodes

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

package Node$n is
    type OPER is (PS段时间1, PS段时间2, ..., PS段时间k);  -- Op Codes for Remote Services
    -- other type definitions ...
    Host: constant NODE := N時間n;  -- remote node
    Conn: constant NSET := (...=> True, others => False);  -- other constant declarations ...

generic
    Site: in NODE;
    package Service is
        procedure PS1 (...);
        ... procedure PSk (...);
    end Service;
end Node$n;
...

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 N时段1, N时段2, ..., N时段n, whereas in practice the individual values would correspond to application-specific mnemonic names (e.g., N时段1 might be written as the Ada identifier "FileServer"). Also, PS段时间1, ..., PS段时间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 encoding 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 communications level, namely Transmit call (XC) and Transit 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, ChmDriver and ChmServer. Each of these have 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 XTP is (XC, XR);

type XMIT(T: XTP) 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$1 is separate;
  ...
end;
package body Node$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$N 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$1);
- Definition of both sides of the higher-level protocol required to support such remote calls, namely the driver side (the body of the generic package Service) and the server side (the body of the non-generic package Support);
- Finally, instantiations of the remote services needed to implement the application modules of this node (package Node$u, Node$v, etc.);

separate (Config)
package body Node$N is
pragma SYSTEM(...);

-- Specify local application modules:

package A$1 is
  procedure Q$1(...);
  ...
  procedure Q$2(...);
  end A$1;

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

-- Local (re)definition of services:

procedure P$1(...) renames A$s.$Q$1;
...

-- 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$1;

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

separate (Config,Node$1)
package body A$1 is
...
Node$1.P$1(...)
end A$1;
...

separate (Config,Node$2)
package body A$m is
...
Node$2.P$1(...)
end A$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.

It must be pointed out, however, that the usage of the Ada separate compilation facilities described above, while legitimate in every respect, may nonetheless cause a potential problem in the context of overly "naive" implementations of those facilities. Specifically, the issue arises in conjunction with circular dependencies (wherein Node$1 calls Node$2, and so must instantiate its Service package which is defined in the body of Node$2, and vice versa). Whereas this, too, could be "programmed around" (at the cost of considerable effort and obscurity), in this instance it would seem preferable to wait for more mature implementations.

4. Possible Protocols

In this section, we shall be concerned with possible protocols by which the desired interprocessor procedure call capability might be 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$1 (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$1. This involves no loss of generality, since the situation 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 Fig. 4-1, namely to permit application processes such as A1, A2, B...C, residing on separate (virtual) machines, to invoke the operation P$1 hosted by Node$1 (corresponding to yet another such virtual machine) as though by a simple (local) procedure call.

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

procedure P$1 (A1: in T$1; ...; A$k: in T$k; R1: out T$1; ...; R$k: out T$k); where A$j stands out for the jth input argument (of type T$aj) and R$k stands for the kth output result (of type T$ak); formal parameters of mode "in" 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 apparently 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 programs at a time. Such an implementation would be 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 might be adopted 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).

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. Driver Side. The overall structure and associated data-flow for the driver side are depicted in Fig. 4-2. Calls on the operation $S_1$, originating from application tasks $T_a...T_z$ are fielded by an Agent which is specific to that operation (AGT1); 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$e$, ($S_1...S_k$) 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$e$. 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.

![Diagram](image)

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

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

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

  task DRH is
    entry DelResp(R: in RESP);
    entry RR$1(...);
    ...
    entry RR$1($1: out Tr1; ...; Ry: out Try);
    ...
    entry RR$k(...);
  end;

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

package D$1 is ... end;
...

package D$k is
  procedure P(Al: in TA1; ...; Ax: in TAx;
              Rl: out Tr1; ...; Ry: out Try);
  procedure PutArg(B: in out BUFF;
                    Al: in TA1; ...; Ax: in TAx);
  procedure GetResp(B: in out BUFF;
                    Rl: out Tr1; ...; Ry: out Try);
end D$k;
...

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, DSI1...DSk, which provides an operation $\Psi_i$ to be called by an application process (as $\Psi_i$) along with operations for moving arguments into and results out of the actual transmission buffers.

The definition of the Driver Call Handler is as follows:

```plaintext
task body DCH is begin loop 
  accept ReqCall(C; in out CALL) do select
    accept DCS1(...) do ... end;
  or accept DCSi(AL: in TA1; ...; AX: in TAX) do
    C.X.Code := OPER'POS(OP$i);
    DSI.PutArg(C.B, AL, ...; AX); 
    end DCSi;
  or accept DCSk(...) do ... end;
  end select;
  end ReqCall;
end loop;
end DCH;
```

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 a particular operation (entry DCSi), whereupon it sets the Opcode of the transaction record for that CALL and transfers the arguments into the associated data buffer.

The definition of the Server Response Handler shows the other side of this interface with the Local Channel Driver for Nodek:

```plaintext
task body DRH is begin loop 
  accept DelResp(R; in RESP) do case OPER'VAL(R.X.Code) is
    when OP$i => ...;
    ... when OP$i => accept RS$i(RL: out TRl, ...; 
      Ry: out TRY) do
      DSI.GetRes(B, RL, ...; Ry); 
    end RS$i;
    end case;
  end DelResp;
end loop;
end DRH;
```

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 RS$i), transferring the corresponding result data.

The outline of the body for a Driver package is shown below:

```plaintext
package body D$i is
  task ACT is entry Exec(AL: in TA1; ...; AX: in TAX; 
    RL: out TR1; ...; Ry: out TRY) 
      renames ACT.Exec;
    procedure PutArg(...) is ... end;
    procedure GetRes(...) is ... end;
    ... + body of AGT 
  end D$i;
```

The (sole) Agent for the operation $\Psi_i$ is simply defined as a task having an entry Exec (with the same signature), and the operation is renamed to be a call to this entry (which is sufficient to ensure the desired property—that calls from the application tasks of each node will be serviced sequentially). In addition, the low-level operations PutArg and GetRes are defined herein (presumably in terms of representation specifications and/or untyped conversions).

Finally the body of the agent task for $\Psi_i$ is defined as follows:

```plaintext
task body AGT is begin loop 
  accept Exec(AL: in TA1; ...; AX: in TAX; 
    RL: out TR1; ...; Ry: out TRY) do
    DCH.DCSi(AL, ...; AX);
    DRH.RS$i(RL, ...; Ry);
    end Exec;
  end loop;
end AGT;
```

For each successive external call to the entry Exec (while the calling process is held in rendezvous), the Agent first delivers the call to DCH and then requests the response from DRH. Because these transactions take place within the rendezvous itself, arguments and results need only be copied once (via the operations PutArg and GetRes) upon actual transmission.

4.1.2. The Server Side. 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 Fig. 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 $\Psi_i$ are performed by one of a
number of surrogate processes (SCTi), 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...An supported by Nodeh).

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

package body Support is
  -- Server Side, defined in Config.Nodeh;
  task SCH is
    entry DelCall(C: in CALL);
    entry RC$i(...);
    entry RC$k(...);
    end;
  task SHh is
    entry Req$Resp(R: in out RESP);
    entry DR$i(...);
    entry DR$k(...);
    end;
  package LCS is new ChnServer(
    From => Conn, To => Host,
    Deliver => SCH.DelCall,
    Request => SHh.Req$Resp);
  package SS$i is ... end;
  package SS$k is
    procedure Go$Arg(B: in BUFF; A: out TAl; ...,
       Ax: out TAx);
    procedure Pu$Res(B: in out BUFF;
       Rl: in TAl; ...,
       Ry: in T Ry);
    end;
  package SHh is ... end;
  + bodies of SCH, SHh, SS$i, ..., SS$k
end Support;

The handler processes are again directly specified as Ada tasks (SCH and SHh) 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 SS$i...SS$k are introduced here for each individual operation PS$i...PS$k that can be called remotely.

The definition of the Server Call Handler is as follows:

task body SCH is
begin loop
  accept DelCall(C: in CALL) do
    case OPER$VAL(C.X.Code) is
      when OP$1 = ...;
      ... accept RC$i(XR: out XREC; Al: out TAl; ...;
                        Ax: out TAx) do
          XR := C.X;
          SS$i.Put$Arg(C.B, Al, ..., Ax);
        end RC$i;
      ... when OP$K = ...;
      end case;
      end DelCall;
    end loop;
  end SCH;
end;

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$i) from the next of the (possibly many) surrogates which are queued up on the corresponding entry. This dispatching consists simply of copying the transaction record contained within this particular CALL and transferring the associated arguments (via the operation Put$Arg provided by SS$i).

The definition of the Server Response Handler is like that of the Call Handler on the driver side:

task body SHh is
begin loop
  accept Req$Resp(R: in out RESP) do
    select
      accept DR$i(...)
    do ... end;
    or
      accept DR$k(XR: in XREC; Rl: in TAl; ...;
                  RY: in T Ry) do
        RX := XR;
        Put$Res(R.B, Rl, ..., Ry);
      end DR$i;
    or
      accept DR$k(...)
    do ... end;
    end select;
    end Req$Resp;
  end loop;
  end SHh;

Each time LCS requests a new response (entry Req$Resp), SHh makes an arbitrary choice among pending responses ready to be delivered for any operation (entries DR$i...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 ori-
ginated.

The definition of a Server package $S$ has 
the following form:

package body $S$ is

subtype SID is NATURAL range 1..Card(Conn);

task type SCT;

ST: array (SID) of SCT; -- surrogate tasks

procedure GetArg(...) is ... end;

procedure PutRes(...) is ... end;

... + body of SCT

derend $S$;

The Surrogates for the operation $P$ are introduced 
as an array of tasks, the range of which is set 
to the cardinality of the incoming connections 
(which would be the maximum number needed if 
every connected node did indeed call the operation 
in question). The operations GetArg and PutRes are 
prematurely the inverses of PutArg and GetRes, 
which were present on the driver side.

Finally, each individual surrogate for $P$ is 
defined as follows:


task body SCT is

XR: XREC;

Al: TAI

... Ax: Tax;

Rk: TRk;

... 

Rx: Try;

begin 

loop

SCH.REC$1(XR, Al, ..., Ax);

Node.$k$.AI..., AX, Rk, ..., Rx);

end loop;

derend SCT;

In a cyclic fashion they simply request a call 
from SCH, invoke the local operation provided by 
Node$, and deliver the corresponding response 
(along with the original transaction record) to be 
dispatched by SCH. Once again, because the dis-
patching is handled within a rendezvous, informa-
tion is copied directly between the individual 
Surrogates and an incoming CALL or outgoing RESP.

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 
side. The servers simply invoke the local oper-
ations in question. If these have been specified 
as entries, then those calls will indeed be serv-
iced sequentially; otherwise they will proceed con-
currently.

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 load 
balancing, is fundamental to the solutions devel-
oped 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 be 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 solu-
tion). This somewhat more general strategy is 
described as a modification to the approach devel-
oped initially.

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

package Config is

type NODE is (NN1, NN2, ..., NNn); 
type NSET is array (NODE) of BOOLEAN;

subtype CONC is INTEGER range 0....;

derend Config;

package Node$1 is ... end; -- Max Concurrency ...

package Node$k is

type OPER is (OP1, OP2, ..., OP$k);

type MFLX is array (OPER) of CONC;

... other type definitions ...

Host: constant NODE := NN$k;

Conn: constant NSET := (... => True, others 

=> False);

Load: constant MFLX := ...;

... other constant declarations ...

generic

Site: in NODE;

Usag: in MFLX;

package Service is

procedure P$1 (...);

... procedure P$k (...);

derend Service;

end Node$k;

... package Node$k is ... end;

derend 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 Nodep, 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 Nodep, 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 XTPY is (XC, XR);
type XMT(T: XTPY) is record
  X: XREC;
  B: BUFF;
end record;

type CALL is XMT(XC);
subtype RESP is XMT(XR);

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

generic
  From: in NODE;
  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$1 is separate;
...
package body Node$m 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$# is

pragma SYSTEM(...);

-- Specify local application modules:
package A$1 is
  procedure Q$1(...);
  ...
  procedure Q$6(...);
end A$1
...
package A$m is
  procedure Q$i(...);
  ...
  procedure Q$s(...);
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$#;

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 additional ID parameter to the DC$1 entries of the Driver Call Handler (DCH), and of introducing a Post Response procedure (PR) to each of the Driver Package DC$1...DC$k;
package body Service is
  -- Driver Side, defined in Config.NodeSh:

task DCH is
  entry ReqCall(C: in out CALL); enter DC$1(...); ...
  entry DC$1(ID: in RCID; Al: in TAL; ...; Ax: in TAX);
  ...
  entry DC$2(...); end;

task DRH is
  entry DelResp(R: in RESP); enter RR$1(...);
  ...
  entry RR$1(RL: out TR1; ...; RY: out TRY); ...
  entry RR$2(...); end;

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

package D$1 is ...
end ...

package D$1 is
  procedure P(Al: in TAL; ...; Ax: in TAX;
  RL: out TR1; ...; RY: out TRY)
  procedure PutArg(B: in outBUFF;
  Al: in TAL; ...; Ax: in TAX);
  procedure GetRes(B: in outBUFF; RL: out TR1; ...;
  RY: out TRY)
  procedure PR(ID: in RCID)
end D$1;
...

package D$2 is ...
end ...

package P$1 (...) renames D$1.P;
...

package P$2 (...) renames D$2.P;
...

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:

task body DCH is
  begin loop
    accept ReqCall(C: in out CALL) do
      select
        accept DC$1(...); do ... end;
      ...
      or
        accept DC$1(ID: in RCID; Al: in TAL; ...;
        Ax: in TAX) do
        C.X.Code := OPER'POS(OP$1);
        C.X.Iden := ID;
        D$1.PutArg(C.B, Al, ...; Ax);
        end DC$1;
      ...
      or
        accept DC$2(...); do ... end;
      ... end select;
      end ReqCall;
  end loop;
end DCH;

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:

task body DRH is
  begin loop
    accept DelResp(R: in RESP) do
      case OPER'VAL(R.X.Code) is
        when OP$1 => ...;
        ...
        when OP$2 =>
          D$1.PR(R.X.Iden);
          accept RR$1(RL: out TR1; ...;
          RY: out TRY) do
          D$1.GetRes(R.B, RL; ...; RY);
          end RR$1;
        ...
        when OP$3 => ...;
      end case;
      end DelResp;
  end loop;
end DRH;

Within a Driver package D$1, the modifications consist primarily of introducing a multiplicity of Agents for the same operation (whereas there was only one heretofore). As shown on the next page, 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 PS1. At the same time, additional entries have to be provided for each call task: these are Init (wherby an Agent acquires its own identity) and Done (whereby it may be notified that the response for the call 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 PS1.

package body D$1 is
  subtype AID is RCID range 1..Usag(OP$1);

  task type AGT is
    entry Init(A: in AID);
    entry Exec(Al: in TAL; ...; Ax: in TAX;
    RL: out TR1; ...; RY: out TRY);
    entry Done;
  end;

  AT: array(AID) of AGT;

48
task AM is
  entry Ready(A; out AID);
  entry Avail(ID; in AID);
end;

procedure P(Al: in TA1; ...; Ax: in TAx;
           R1: out TR1; ...; Ry: out TRY) is
  A: AID;
begin
  AM, Ready(A);
  AT(A, Exec(Pl, ...; Ax, R1; ...; Ry));
end;

procedure PutArg(...) is ... end;
procedure GetArg(...) is ... end;

procedure PR(ID: in NCID) is
begin
  AT(AID; ID).Done;
end;

... + bodies of ACT, AM
end DSI;

The initialization and actual allocation of agents is handled by the Agent Manager:

task body AM is
begin
  for A in AID loop
    AT(A).Init(A);
  end loop;

  -- main cycle:
  loop
    accept Ready(A; out AID) do
      accept Avail(ID; in AID) do
        A := ID;
      end;
    end;
  end loop;
end AM;

Each of the agent tasks of the array AT is then defined as follows:

task body ACT is
begin
  accept Init(A; in AID) do
    ID := A;
  end;

  -- main cycle:
  loop
    AM.Avail(ID);
    accept Exec(Al: in TA1; ...; Ax: in TAx;
               R1: out TR1; ...; Ry: out TRY) do
        DCH.DCSI(1D; Al; ...; Ax);
        accept Done;
        DRR.BRSL(E; ...; Ry);
    end Exec;
  end loop;
end ACT;

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 PSI, which would be specified within the corresponding Server package SSI as follows:

- subtype SID is CONC range 1..Load (OPSL);
- 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 PSI) upon which both of the protocols developed in this section have been based.

6. Conclusion

This paper has addressed the problem of programming distributed applications in Ada and outlined a first approach in this area. Essentially two aspects have been considered: the provision of a suitable compile-time framework for defining such applications in the first place (which was achieved by exploiting the possibilities of the separate compilation facilities in Ada); and the support of a suitable "Interprocess procedure call" protocol, whereby the application itself could then be programmed without further regard to the distributed nature of the underlying hardware configuration (a capability which was defined in terms of the multi-tasking facilities of Ada).

Several such protocols were in fact developed here, beginning with the relatively simple Remote Entry Call, which was then extended to yield the Remote Procedure Call strategy. In [4] we 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 were perfectly reliable.

References


