CMU-CS-85-168

Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing

Barbara Liskov
M.I.T. Laboratory for Computer Science
Maurice Herlihy
C.M.U. Computer Science Department
Lucy Gilbert
Autographix, Inc.

15 October 1985

Abstract

Modules in a distributed program are active, communicating entities. A language for distributed programs must choose a set of communication primitives and a structure for processes. This paper examines one possible choice: synchronous communication primitives (such as rendez-vous or remote procedure call) in combination with modules that encompass a fixed number of processes (such as Ada tasks or UNIX processes). An analysis of the concurrency requirements of distributed programs suggests that this combination imposes complex and indirect solutions to common problems and thus is poorly suited for applications such as distributed programs in which concurrency is important. To provide adequate expressive power, a language for distributed programs should abandon either synchronous communication primitives or the static process structure.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

1. Introduction

A distributed system consists of multiple computers (called nodes) that communicate through a network. A programming language for distributed computing must choose a set of communication primitives and a structure for processes. In this paper, we examine one possible choice: synchronous communication primitives (such as rendez-vous or remote procedure call) in combination with modules that encompass a fixed number of processes (such as Ada tasks or UNIX processes). We describe and evaluate the program structures needed to manage certain common concurrency control problems. We are concerned here not with computational power, but with expressive power: the degree to which common problems may be solved in a straightforward and efficient manner. We conclude that the combination of synchronous communication with static process structure imposes complex and indirect solutions, and therefore that it is poorly suited for applications such as distributed programs in which concurrency is important. To provide adequate expressive power, a language for distributed programming should abandon either synchronous communication primitives or the static process structure.

Our analysis is based on the *client/server model*, in which a distributed program is organized as a collection of modules, each of which resides at a single node in the network. Modules do not share data directly; instead they communicate through messages. Modules act as clients and as servers. A *client* module makes use of services provided by other modules, while a *server* module provides services to others by encapsulating a resource, providing synchronization, protection, and crash recovery. The client/server model is hierarchical: a particular module may be both a client and a server.

Although other models for concurrent computation have been proposed, the hierarchical client/server model has come to be the standard model for structuring distributed programs. Lauer and Needham [16] argued that the client/server model is equivalent (with respect to expressive power) to a model of computation in which modules communicate through shared data. Nevertheless, the client/server model is more appropriate for distributed systems because speed and bandwidth are typically more critical in the connection between a module and its local data than between distinct modules. Although certain specialized applications may fit naturally into alternative structures such as pipelines or distributed coroutines, the hierarchical client/server model encompasses a large class of distributed programs.

This paper is organized as follows. In Section 2, we propose a taxonomy for concurrent systems,

focusing on primitives for intermodule communication and the process structure of modules. In Section 3, we formulate the basic concurrency requirement for modules: if one activity within a module becomes blocked, other activities should be able to make progress. In Section 4, we describe and evaluate the program structures needed to satisfy the concurrency requirement using synchronous communication primitives with a static process structure. We conclude that the only reasonable solution to certain problems requires using the available primitives to simulate asynchronous communication primitives and/or a dynamic process structure. In Section 5, we summarize our results and make suggestions about linguistic support for distributed computing.

2. Choices for Communication and Process Structure

In this section we review the range of choices for communication and process structure and identify the choices made in various languages.

2.1. Communication

There are two main alternatives for communication primitives: synchronous and asynchronous. *Synchronous* mechanisms provide a single primitive for sending a request and receiving the associated response. The client's process is blocked until the server's response is received. Examples of synchronous mechanisms include procedure call, remote procedure call [21], and rendez-vous [7]. Languages that use synchronous mechanisms for communication include Mesa [20], DP [5], Ada [7], SR [1],[1] MP [24], and Argus [18].

Asynchronous communication mechanisms typically take the form of distinct *send* and *receive* primitives for originating requests and acquiring responses. A client process executing a send is blocked either until the request is constructed, or until the message is delivered (as in CSP [13]). The client acquires the response by executing the receive primitive. After executing a send and before executing the receive, the client may undertake other activity, perhaps executing other sends and receives. Languages that use send/receive include CSP and PLITS [8].

---

[1] In addition to remote call, SR provides the ability to send a request without waiting for the response.

2.2. Process Structure

There are two choices for process structure within a module. Modules having a *static* structure encompass a fixed number of threads of control (usually one). The programmer is responsible for multiplexing these threads among a varying number of activities. Examples of modules having a static process structure include Ada tasks, DP monitors, and CSP processes, where there is just one process per module, and SR, where there may be multiple processes per module. The multiplexing mechanisms available to the programmer include guarded commands and condition variables.

An alternative to the static structure is the *dynamic* structure, in which a variable number of processes may execute within a module. (Note that a module's process structure is independent of the encompassing system's process structure; the number of processes executing within a module may vary dynamically even if the overall number of processes in the system is fixed.) The system is responsible for scheduling, but the programmer must synchronize the use of shared data. Ada, MP, Argus, and Mesa are examples of languages in which the basic modular unit encompasses a dynamic process structure.

2.3. Combinations

All four combinations of communication and process structure are possible. Figure 2-1 shows the combinations provided by several languages. In Argus, the basic modular unit is the *guardian*, which has a dynamic process structure. In MP, a language for programming highly parallel machines, shared data are encapsulated by *frames*, which encompass a dynamic process structure. In CSP, the process itself is the basic modular unit. A *monitor* in DP contains a single thread of control that is implicitly multiplexed among multiple activities. A *resource* in SR can contain a fixed number of processes. In Mesa monitors, the processes executing in *external* procedures have a dynamic structure. Access to shared data is synchronized through *entry* procedures that acquire the monitor lock. In Starmod [6], a dynamic process model with synchronous communication can be obtained through the use of *processes*; a static process model with asynchronous primitives can be obtained through the use of *ports*.

In Ada, a *task* is a module with a static process structure. A server task exports a collection of entries, which are called directly by clients. A server task's process structure is static because a task encompasses a single thread of control. (Although a task can create subsidiary tasks dynamically, these subsidiary tasks cannot be addressed as a group.) The programmer of the server task uses

select statements to multiplex the task among various clients. Note that a *task family* (an array of tasks) does not qualify as a module, because the entire family cannot be the target of a call; instead a call must be made to a particular member of a family.

Alternatively, a distributed Ada program might be organized as a collection of *packages* that communicate through interface procedures. This alternative, however, can be rejected as violating the intent of the Ada designers. The Ada Rationale [14] explicitly states that internode communication is by entry calls to tasks, not by procedure calls to packages.[2] This alternative can also be rejected on technical grounds; we argue in Section 4.3 that certain extensions to Ada are necessary before packages can be used effectively as modules in distributed programs.

We are not aware of any languages that provide asynchronous communication with dynamic processes. Although such languages may exist, this combination appears to provide an embarrassment of riches not needed for expressive power.

	Static	*Dynamic*
Synchronous	Ada Tasks, DP, SR	Argus, Mesa, Starmod, MP
Asynchronous	CSP, PLITS, Starmod	

Figure 2-1: Communication and Process Structure in Some Languages

3. Concurrency Requirements

In this section we discuss the concurrency requirements of the modules that make up a distributed program. Our discussion centers on the concurrency needs of modules that act as both clients and servers; any linguistic mechanism that provides adequate concurrency for such a module will also provide adequate concurrency for a module that acts only as a client or only as a server.

The principal concurrency requirement is the following: if one activity within a module becomes blocked, other activities should be able to make progress. A system in which modules are unable to

---

[2] "Note ... that on distributed systems (where tasks do not share a common store) communication by procedure calls may be disallowed, all communication being achieved by entry calls." (Page 11-40).

set aside blocked activities may suffer from unnecessary deadlocks and low throughput. For example, suppose a server cannot carry out one client's request because another client has locked a needed resource. If the server then becomes blocked, rendering it unable to accept a request to release the resource, then a deadlock will occur that was otherwise avoidable. Even when there is no prospect of deadlock, a module that remains idle when there is work to be done is a performance bottleneck.

We can distinguish two common situations in which an activity within a module might be blocked:

1. *Local Delay:* A local resource needed by the current activity is found to be unavailable. For example, a file server may discover that the request on which it is working must read a file that is currently open for writing by another client. In this situation, the file server should temporarily set aside the blocked activity, turning its attention to requests from other clients.

2. *Remote Delay:* The module makes a call to another module, where a delay is encountered. The delay may simply be the communication delay, which can be large in some networks. Alternatively, the delay may occur because the called module is busy with another request or must perform considerable computing in response to the request. While the calling module is waiting for a response, it should be able to work on other activities.

We do not discuss the use of I/O devices in our analysis below, but it is worth noting that remote delays are really I/O delays, and remarks about one are equally applicable to the other.

4. Program Structures

We now discuss several program structures that might be used to meet the concurrency requirement stated above. First, we review several techniques used to make progress in the presence of local delays. We conclude that the synchronous communication/static process combination provides adequate expressive power for coping with local delays (although the particular mechanisms provided by Ada are awkward for this purpose). In the second part, however, we argue that this combination provides inadequate expressive power for coping with remote delays.

In the following discussion, examples are shown as Ada tasks. We chose Ada because we assume it will be familiar to many readers. Although we call attention to some problems with Ada, this paper is not intended to be an exhaustive analysis of the suitability of Ada for distributed computing (see, however, [23, 9, 15]). Instead, the examples are intended to illustrate language-independent problems with the synchronous communication/static process combination.

4.1. Local Delay

Our evaluation of techniques for coping with local delays is based on work of Bloom [4], who argued that a synchronization mechanism provides adequate expressive power to cope with local delays only if it permits scheduling decisions to be made on the basis of the following information:

- the name of the called operation,
- the order in which requests are received,
- the arguments to the call, and
- the state of the resource.

We use these criteria to evaluate techniques for avoiding local and remote delays.

Languages combining static process structure with synchronous communication provide two distinct mechanisms for making progress in the presence of local delay:

1. *Conditional Wait.* The conditional wait is the method used in monitors. An activity that encounters a local delay relinquishes the monitor lock and waits on a queue. While that activity is suspended, other activities can make progress after acquiring the monitor lock. Conditional wait is quite adequate for coping with local delays because scheduling decisions can employ information from all the categories listed above.

2. *Avoidance.* Avoidance is used in languages such as Ada and SR. Boolean expressions called *guards* are used to choose the next request. A server will accept a call only after it has ascertained that the local resources needed by the new request are currently available. Avoidance is adequate for coping with local delay as long as the guard mechanism is sufficiently powerful. In SR, for example, guards may employ information from all of Bloom's categories. In Ada, however, guards for **accept** statements cannot depend on the values of arguments to the call, and therefore the Ada guarded command mechanism lacks the expressive power needed to provide a general solution to the problem of local delays.

.The shortcomings of the Ada guarded command mechanism can be illustrated by the following simple example, which will be used throughout the paper. A *Disk Scheduler* module synchronizes access to a disk through REQUEST and RELEASE operations (c.f. [12, 4]). Before reading or writing to the disk, a client calls REQUEST to identify the desired track. When REQUEST returns, the client accesses the disk and calls RELEASE when the access is complete. The scheduler attempts to minimize changes in the direction the disk head is moving. A request for a track is postponed if the track does not lie in the head's current direction of motion. The direction is reversed when there are

no more pending requests in that direction.

Perhaps the most natural implementation of the disk scheduler server in Ada would provide two entries:

```
task DISK_SCHEDULER  is
        entry REQUEST(ID: in TRACK);
        entry RELEASE;
end DISK_SCHEDULER
```

Internally, the server keeps track of the current direction of motion, and the current head position:

```
task body DISK_SCHEDULER is
        type STATUS = (UP,DOWN,NEUTRAL);
        -- current head position.
        POSITION: TRACK;
        -- current direction of motion.
        DIRECTION: STATUS;
        ...
end DISK_SCHEDULER
```

A naive attempt at scheduling is:

```
select -- Warning: not legal Ada!
  when (DIRECTION = UP and POSITION <= ID) =>
    -- Move up.
    accept REQUEST(ID: in TRACK) do ...
or  when (DIRECTION = DOWN and POSITION >= ID) =>
    -- Move down.
    accept REQUEST(ID: in TRACK) do ...
or  when (DIRECTION = NEUTRAL) =>
    -- Move either way.
    accept REQUEST(ID: in TRACK) do ...
```

This program fragment is illegal, however, because Ada does not permit an argument value (i.e., ID) to be used in a **when** clause. As a consequence of this restriction, modules whose scheduling policies depend on argument values must be implemented in a more roundabout manner.

In the remainder of this section, we use the disk scheduler example to illustrate five alternative techniques for avoiding local delay. We argue that although some of these techniques will help in special cases, only one of them, the Early Reply structure, is powerful enough to provide a fully general solution to the local delay problem.

1. *Entry Families.* It is possible to treat an argument specially by using an indexed *family* of entries. If the index is viewed as an argument to the call, entry families provide a limited ability to accept requests based on arguments.

For example, REQUEST could be implemented as a family containing an entry for each track. The

index provides an indirect way to incorporate the track number into the entry name, and so, for example, the task can delay accepting a request to write to a track that lies in the wrong direction from the disk head.

```
case DIRECTION is
  when UP =>
    for I in POSITION..TRACK'LAST loop
      accept REQUEST(I) do ...
    end loop;
  when DOWN =>
    for I in reverse POSITION..TRACK'FIRST loop
      accept REQUEST(I) do ...
    end loop;
  when NEUTRAL =>
    for I in TRACK'RANGE loop
      accept REQUEST(I) do ...
    end loop;
end case
```

A source of awkwardness here is that the **accept** statement can refer only to values of local variables (I in this case). The first two arms of the **case** statement must poll each of the entries in turn, and in the third arm, the entire family must either be polled sequentially as shown, or each track must have its own **accept** statement. Neither choice is attractive, particularly if the number of tracks is large. Similar problems have been noted by [27].

Perhaps a more important limitation of entry families is that they are effective only when the subsumed argument's type is a discrete range. For example, entry families could not be used if the range of disk tracks were to vary dynamically, or if requests were granted on the basis of a numeric priority, as in the "shortest job first" scheduler of [26].

2. *Refusal.* A server that accepts a request it is unable to service might return an exception to the client to indicate that the client should try again later.

Refusal can eliminate local delays, but it is an awkward and potentially inefficient technique. When a client receives such an exception, it must decide whether to try again or to pass the exception to a higher level. If it tries again, it must decide when to do so, and how many times to retry before giving up. Note that such decisions present a modularity problem, because they are likely to change as the underlying system changes. For example, if a server's load increases over time, it will refuse more requests, and its clients will be forced to adjust their timeout and retry strategies. The server might also have to roll back work already done on the client's behalf, work that may have to be redone later, perhaps several times.

3. *Nested Accept*. A server that encounters a local delay might accept another request in a nested **accept** statement.

The following code fragment shows an unlikely way of programming a disk scheduler. The server indiscriminately accepts calls to REQUEST. When it discovers that it has accepted a request for a track that lies in the wrong direction, it uses a nested **accept** statement to accept a different request. When the disk head direction changes, the server resumes processing the original request.

```
accept REQUEST(ID: in TRACK) do
  -- If the direction is wrong,
  -- then accept something else.
  while DIRECTION = UP and ID < POSITION loop
    select
      accept REQUEST(ANOTHER_ID: in TRACK) do
      ...
    or
      terminate;
    end select;
  end loop;
  ...
end REQUEST;
```

At first glance the nested **accept** technique might appear similar to the conditional wait technique supported by monitors, in which an activity that is unable to make progress is temporarily set aside in favor of another activity. The critical difference between the two techniques is that the use of nested **accept** statements requires that the new call be completed before the old call can be completed, an ordering that may not correspond to the needs of applications. Monitors do not impose the same last-in-first-out ordering on activities. A second difficulty with nested **accept** statements is deciding which request to accept. For example, the disk scheduler will be no better off if it accepts another request for a track that lies in the wrong direction. Note that if it were possible to decide which requests to accept after getting into trouble, then it should have been possible to accept only non-troublesome requests in the first place, thus avoiding the use of the nested **accept**. This observation suggests that the nested **accept** provides no expressive power that is not otherwise available.

4. *Task Families*. Instead of using a single task to implement a server, one might use a *family* (i.e., an array) of identical tasks.[3] These tasks would together constitute the server, and would synchronize with one another in their use of the server's data. If one task encounters a local or remote delay, the server need not remain idle if another task is still

[3]DP and SR provide analogous mechanisms.

able to service requests.

The principal difficulty with task families lies in allocating tasks among clients. As mentioned above, a client cannot make a call to a task family as a whole; instead it is necessary to allocate family members to the clients in advance of the call. If the server's clients can be identified in advance, then it is possible to have a task family member for each client, and each client can be permanently assigned its own private task. This structure can avoid local delays, but such static assignments are not feasible for many applications.

If static allocation is not possible, then two alternative methods might be used. A client could choose a task on its own, perhaps at random, and then use a timed or conditional entry call to determine if it is free. If the task is not free, the client could try again. Such a method is cheap if contention for tasks in the family is low, but it can result in arbitrary delays when contention is high.

Alternatively, the server could provide a manager task that assigns tasks to clients. Although the manager can avoid conflicts in the use of family members, extra messages are needed when a task is allocated or freed. A manager is expensive if a task is used just once for each allocation; the cost decreases as the number of uses per allocation increases.

In either the static or dynamic case, task families require enough family members to make the probability of contention acceptably low. Since most tasks in the family are likely to be idle at any time, the expense of the technique depends on how cheaply idle tasks can be implemented.

If the number of clients is static, task families do provide a solution to local delay. Nevertheless, they do not provide a general solution because often the number of clients is dynamic (or unknown). In this case, task families can at best provide a probabilistic guarantee against delay.

5. *Early Reply*. A fully general solution to the concurrency problem can be constructed if synchronous primitives are used to simulate asynchronous primitives.

A server that accepts a request from a client simply records the request and returns an immediate acknowledgment. The server carries out the request at some later time, and conveys the result to the client through a distinct exchange of messages. Numerous examples employing Early Reply appear in the literature (e.g. see [10, 26, 25]).

A disk scheduler task employing Early Reply would export three entries: a REGISTER entry, a REQUEST entry family, and a RELEASE entry. The client calls REGISTER to indicate the disk track desired. The server records the request and responds with an index into the REQUEST entry family.

```
entry REGISTER(ID: in TRACK, CLIENT_ID: out INDEX);
```

An outline of a disk scheduler server employing early reply to avoid local delays appears in Figure 4-1. Although early reply provides a general technique for avoiding local delays, it is potentially inefficient and awkward. Efficiency may suffer because an additional remote call is needed whenever a delay is possible (e.g. when requesting a disk track). Program clarity may suffer because the server must perform its own scheduling, explicitly multiplexing its thread of control among multiple clients.

When the server is ready to grant a client's request, it accepts a call to that client's member of the REQUEST entry family. For this structure to work, the client must follow a protocol in which it first registers its request, and then makes a second call to receive its response. To avoid delaying the server, the client should follow the REGISTER call with an immediate call to REQUEST:

```
SERVER.REGISTER(TRACK_ID, MY_ID);
SERVER.REQUEST(MY_ID);
```

Most of the techniques discussed in this paper, including Early Reply, work best if clients access server tasks indirectly through procedures exported by a package local to the client's site. The package enhances safety by enforcing protocols and by ensuring that identifiers issued to clients are not misused.

4.2. Remote Delay

We now go on to discuss the problem of remote delays. We do not claim that it is impossible to avoid remote delay under the synchronous communication/static process structure combination. We argue instead that this combination imposes complex and indirect solutions to common problems that arise in distributed programs. To illustrate this point, we review the techniques discussed above for coping with local delay. The only technique that provides a general solution to the remote delay problem is Early Reply. We focus on two servers: the *higher-level* server called by a client, and a *lower-level* server called by the higher-level server. Most of the techniques previously considered for coping with local delay are clearly inadequate for coping with remote delay:

- *Conditional Wait.* The inability of monitors to cope with remote delay is known as the *nested monitor call* problem [19, 11]. When a monitor makes a call to another monitor, the calling activity retains the monitor lock, and the monitor must remain idle while the call is in progress.

- *Avoidance.* Remote delays cannot be eliminated by avoidance. A server cannot choose to avoid a client's request just because it may lead to a remote delay. The lower-level

```
task DISK_SCHEDULER is
  ...
  entry REGISTER(ID: in TRACK; CLIENT_ID: out INDEX);
  entry RESPONSE(INDEX);
  entry RELEASE;
  ...
end SERVER;
task body SERVER is
  ...
begin
  loop
    BUSY := false;
    NEXT_CLIENT := UNKNOWN;
    -- Accept and enqueue new requests.
    select
      accept REGISTER(ID: in TRACK; out CLIENT_ID: INDEX) do
        -- Allocate and return index.
        CLIENT_ID := ...;
      end REGISTER;
      ... -- Enqueue request and client id.
      -- compute next client and next track.
      NEXT_CLIENT := ...
      NEXT_TRACK := ...
    or
      when not BUSY and NEXT_CLIENT /= UNKNOWN =>
        accept REQUEST(NEXT_CLIENT);
        POSITION := NEXT_TRACK;
        BUSY := true;
    or
      when BUSY =>
        accept RELEASE;
        BUSY := false;
        ... -- Change direction if necessary.
        NEXT_CLIENT := ...
        NEXT_TRACK := ...
      end select;
    end loop.
  end loop;
end SERVER;
```

Figure 4-1: Using Early Reply to Avoid Local Delays

server could avoid accepting the higher-level server's call if it would otherwise encounter a local delay, but the higher-level server is delayed in the meantime. Avoidance can thus lead to both deadlock and performance problems.

- *Refusal.* Refusal is no more effective for dealing with remote delays than it is for dealing with local delays.

- *Nested Accept.* One might attempt to avoid remote delays through the use of nested **accept** statements in the following way: a task creates a subsidiary task to carry out the

remote call while the creating task executes a nested **accept**. However, the necessity of finishing the nested call before responding to the outer call renders this technique of dubious value.

- *Task Families*. The remarks made about task families when discussing local delays apply to remote delays as well. Task families work well when the number of clients is static and predetermined. Otherwise, task families provide only a probabilistic guarantee against delay. Probabilistic guarantees concerning performance might well suffice for some applications, but probabilistic guarantees concerning deadlock seem less useful.

Only Early Reply provides a general solution to the problem of remote delay, but the efficiency and program complexity problems noted above become more acute. If the lower-level server employs early reply, then the higher-level server will be subject to remote delay if it requests its response prematurely. To avoid waiting, the higher-level server might use periodic timed or conditional entry calls to check for the response. This structure is potentially inefficient, since the higher-level server is making multiple remote calls, and it may also yield complex programs because the higher-level server must explicitly multiplex these calls with other activities. An alternative structure in which the lower-level server returns its response by calling an entry provided by the higher-level server is no better. To avoid delaying the lower-level server, the higher-level server must accept the call in a timely fashion, and the need for explicit multiplexing is unchanged.

These shortcomings of Early Reply can be alleviated by the introduction of dynamic task creation in the higher-level server. In this structure, the higher-level server uses Early Reply to communicate with clients. When the higher-level server receives a request, it creates a subsidiary task (or allocates an existing task) and returns the name of that task to the client. The structure is illustrated in Figure 4-2. The client later calls the subsidiary task to receive the results of its request.

```
SERVER.REQUEST( ..., TASK_ID);
...
TASK_ID.RESPONSE(...);
```

The subsidiary task carries out the client's request, making calls to lower-level servers, and using shared data to synchronize with other subsidiary tasks within the higher-level server. When work on the request is finished, the subsidiary task accepts the client's second call to pick up the response.

This structure avoids the program complexity problems associated with Early Reply. The higher-level server need not perform explicit multiplexing among clients because the tasking mechanism itself multiplexes the processor(s) among the various activities, although, as mentioned above,

subsidiary tasks must synchronize through shared data. This structure also alleviates the need to synchronize with lower-level servers. The server as a whole is not affected if one subsidiary task is delayed, because other subsidiary tasks can still work for other clients.

```
task SERVER is
  ...
  entry REQUEST (..., HELPER: out access HELPER_TASK);
  task type HELPER_TASK is
    -- From server:
    entry FORWARD_REQUEST(...);
    -- From client:
    entry RESPONSE(...);
  end HELPER_TASK;
end SERVER;
task body SERVER is
  ...
  task body HELPER_TASK is
    ...
  begin
    accept FORWARD_REQUEST(...) do
      ... -- Put arguments in local variables.
    end FORWARD_REQUEST;
    ... -- Do work for client,
    ... -- including lower-level calls.
    accept RESPONSE(...) do
      ... -- Return response to client;
    end RESPONSE
  end HELPER_TASK;
begin
  loop
    select
      accept REQUEST(...,
                     HELPER: out access HELPER_TASK)
      do
        HELPER := new HELPER_TASK;
        HELPER.FORWARD_REQUEST(...);
      end REQUEST;
    ...
    or
      terminate;
    end select;
  end loop;
end SERVER;
```

Figure 4-2: Early Reply and Dynamic Task Creation

One disadvantage of this structure is that creating and destroying the server tasks may be

expensive, although this cost can be avoided by reusing the subsidiary tasks.[4] This structure also requires two message exchanges for each client/server interaction. The most striking fact about this mechanism, however, is that we are using synchronous rendez-vous to simulate asynchronous send and receive primitives, and dynamic task creation to simulate a dynamic process structure. The need to use one set of primitives to simulate another suggests that the combination of the static process structure with synchronous communication does not provide adequate expressive power to satisfy the concurrency requirements of distributed programs.

4.3. Remarks about Ada

Although this paper is not intended to be a systematic critique of Ada itself, it does raise some questions about the suitability of Ada for distributed computing. One way to circumvent the limitations of Ada tasks is to use packages instead of tasks as the basic modular unit for constructing servers. A server package would export a collection of interface procedures, which would be called directly by clients. These procedures could synchronize with one another by explicit calls to entries of tasks private to the package. (This structure is similar to a Mesa monitor [20].) A server package would solve the concurrency problem by providing a dynamic process structure; an arbitrary number of client processes can be executing within the package's interface procedures. The programmer is responsible for synchronizing access to shared data, but not for scheduling the processes.

Nevertheless, Ada packages suffer from critical limitations unrelated to the main focus of this paper. In applications such as distributed computing, it is often necessary to store module names in variables, and to pass them as parameters. For example, the Cedar remote procedure call facility [2] uses Grapevine [3] as a registry for server instances and types. Similarly, dynamic reconfiguration, replacing one module with another [15], may be necessary to enhance functionality, to accommodate growth, and to support fault-tolerance. Ada permits names of tasks (i.e. their access values) to be stored in variables and used as parameters, but not names of packages or procedures. Consequently, a distributed program in which the package is the basic modular unit could not support Grapevine-like registries of services or dynamic reconfiguration. These observations suggest that Ada would provide better support for distributed programs if it were

---

[4] In the microVAX-II implementation of Argus, it takes only 160 microseconds to create, run, and destroy a null process.

extended to treat packages as first-class objects.

5. Discussion

We have argued that languages that combine synchronous communication primitives with a static process structure do not provide adequate expressive power for constructing distributed programs. Although our discussion has focused on distributed programs, our conclusions are valid for any concurrent program organized as clients and servers (e.g. interprocess communication in UNIX [22] or programming for highly parallel machines [24]).

Languages that combine synchronous communication with a static process structure can provide adequate expressive power for avoiding *local delays*, in which an activity is blocked because a local resource is unavailable, as illustrated by monitors and by languages with a fully general guarded command mechanism. (Ada's guarded **accept** mechanism, however, lacks expressive power because it does not allow the call's arguments to be used in guards.) The languages under consideration do not, however, provide adequate expressive power for avoiding *remote delays*, in which an activity is blocked pending the completion of a remote call. For monitors, this problem has come to be known as the *nested monitor call* problem [11, 19]. The thesis of this paper is that languages in which modules combine synchronous communication with a static process structure suffer from an analogous problem.

We examined a number of techniques to avoid such delays using Ada tasks. Although certain techniques are adequate for particular applications, e.g. entry families, the only fully general solution requires using synchronous primitives to simulate asynchronous primitives, and dynamic task creation to simulate a dynamic process structure. The need to use one set of primitives to simulate another is evidence that the original set lacks expressive power.

We believe that it is necessary to abandon either synchronous communication or static process structure, but a well-designed language need not abandon both. If the language provides asynchronous communication primitives, in which the sender need not wait for the receiver to accept the message, then the time spent waiting for the early reply in our examples would be saved. In addition, the lower level-server need not be delayed waiting for the higher-level server to pick up the response, and only two messages would be needed. The need to perform explicit multiplexing remains a disadvantage of this choice of primitives.

Alternatively, if a language provides dynamic process creation within a single module, as in Argus

and Mesa, then the advantages of synchronous communication can be retained. When a call message arrives at a module, a new process is created automatically (or allocated from a pool of processes) to carry out the request. When the process has completed the call, a reply message is sent back to the caller, and the process is destroyed or returned to the process pool. Only two messages are needed to carry out the call. The new process has much the same function as the subsidiary tasks created by the server in Figure 4-2. It ensures that the module is not blocked even if the request encounters a delay. The process must synchronize with other processes in the module, but all the processes are defined within a single module, which facilitates reasoning about correctness. Finally, a dynamic process structure can mask delays that result from the use of local input/output devices, and may permit multiprocessor nodes to be used to advantage. We think synchronous communication with dynamic processes is a better choice than asynchronous communication with static processes; a detailed justification for this opinion is given in [17].

A language mechanism cannot be considered to have adequate expressive power if common problems require complex and indirect solutions. Although synchronous communication with static process structure is computationally complete, it is not sufficiently expressive. Synchronous communication is not compatible with the static process structure, and languages for distributed programs must abandon one or the other.

Acknowledgments

The authors would like to thank Beth Bottos, Larry Rudolph, and the members of the Argus design group, especially Paul R. Johnson, Robert Scheifler, and William Weihl for their comments on earlier drafts of this paper.

References

[1] Andrews, G. R.
Synchronizing Resources.
ACM Transactions on Programming Languages and Systems 3(4):405-430, October, 1981.

[2] Birrel A. D., and Nelson, B. J.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[3] Birrel, A. D., Levin, R., Needham, R., and Schroeder, M.
Grapevine: an Exercise in Distributed Computing.
Communications of the ACM 25(14):260-274, April, 1982.

[4] Bloom, T.
Evaluating Synchronization Mechanisms.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*. ACM-SIGOPS, December, 1979.

[5] Brinch Hansen, P.
Distributed processes: A concurrent programming concept.
Communications of the ACM 21(11):934-941, November, 1978.

[6] Cook, R. P.
Starmod- a language for distributed programming.
IEEE Transactions on Software Engineering 6(6):563-571, November, 1980.

[7] Dept. of Defense.
Reference manual for the ADA programming language.
1983.
ANSI/MIL-STD-1815A-1983.

[8] Feldman, J.A.
High Level Programming for Distributed Computing.
Communications of the ACM 22(6):353-368, June, 1979.

[9] Gehani, N.
Concurrency in Ada and Multicomputers.
Computer Languages 7(2), 1982.

[10] Gehani, Narain.
Ada: an advanced introduction.
Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[11] Haddon, B. K.
Nested monitor calls.
ACM Operating Systems Review 11(4):18-23, October, 1977.

[12] Hoare, C.A.R.
Monitors: an operating system structuring concept.
Communications of the ACM 17(10):549-557, October, 1974.

[13] Hoare, C.A.R.
Communicating sequential processes.
Communications of the ACM 21(8):666-677, August, 1978.

[14] Ichbiah, J. et al.
Rationale for the design of the Ada programming language.
SIGPLAN Notices 14(6), June, 1979.

[15] Knight, J. C. and Urquhart, J. I. A.
On the implementation and use of Ada on fault-tolerant distributed systems.
Ada Letters 4(3), November, 1984.

[16] Lauer, P. E., and Needham, R. M.
On the duality of operating systems structures.
In *Proc. Second International Symposium on Operating Systems Structures*. October, 1978.
Reprinted in Operating Systems Review, 13(2) April 1979, pp. 3-19.

[17] B. Liskov and M. Herlihy.
Issues in Process and Communication Structure for Distributed Programs.
In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*. October, 1983.

[18] Liskov, B., and Scheifler, R.
Guardians and actions: linguistic support for robust, distributed programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[19] Lister, A.
The problem of nested monitor calls.
ACM Operating Systems Review 11(2):5-7, July, 1977.

[20] Mitchell, J., G., Maybury, W., and Sweet, R.
Mesa language manual, version 5.0.
Technical Report CSL-79-3, Xerox Palo Alto Research Center, April, 1979.

[21] Nelson, B.
Remote Procedure Call.
Technical Report CMU-CS-81-119, Carnegie-Mellon University, 1981.
Ph.D. Thesis.

[22] Rashid, R.
An inter-process communication facility for Unix.
Technical Report CMU-CS-81-124, Carnegie-Mellon University, 1980.

[23] Schuman, S. A., Clarke, E. M., Nikolaou, C. N.
Programming Distributed Applications in Ada: a First Approach.
In *Proceedings of the 1981 Conference on Parallel Processing*. August, 1981.

[24] Segall, Z. and Rudolph, L.
PIE - a programming and instrumentation environment for parallel processing.
Technical Report CMU-CS-85-128, Carnegie-Mellon University, 1985.

[25] Wegner, P. and Smolka, S.
Processes, tasks, and monitors: a comparative study of concurrent programming primitives.
IEEE Transactions on Software Engineering 9(4):39-59, July, 1983.

[26] Welsh, J., and Lister, A.
A comparative Study of Task Communication in Ada.
Software - Practice and Experience 11:257-290, 1981.

[27] Yemini, S.
On the suitability of Ada multitasking for expressing parallel algorithms.
In *Proceedings of AdaTEC Conference on Ada*. October, 1982.