Chapter 14
Distributed systems

14.1 Distributed system definition
14.2 Overview of issues
14.3 Language support
14.4 Distributed programming systems and environments
14.5 Reliability
14.6 Distributed algorithms
14.7 Deadline scheduling in a distributed environment
Summary
Further reading
Exercises

Over the last thirty years, the cost of microprocessors and communications technology has continually decreased in real terms. This has made distributed computer systems a viable alternative to uniprocessor and centralized systems in many embedded application areas. The potential advantages of distribution include:

- improved performance through the exploitation of parallelism,
- increased availability and reliability through the exploitation of redundancy,
- dispersion of computing power to the locations in which it is used,
- the facility for incremental growth through the addition or enhancement of processors and communications links.

This chapter discusses some of the problems that are introduced when real-time systems are implemented on more than one processor.

14.1 Distributed system definition

For the purposes of this chapter, a **distributed computer system** is defined to be a system of multiple autonomous processing elements cooperating in a common purpose or to achieve a common goal. This definition is wide enough to satisfy most intuitive notions, without descending to details of physical dispersion, means of communication, and so on. The definition excludes pipeline and array processors, whose elements are not autonomous; it also excludes those computer networks (for example, the Internet)

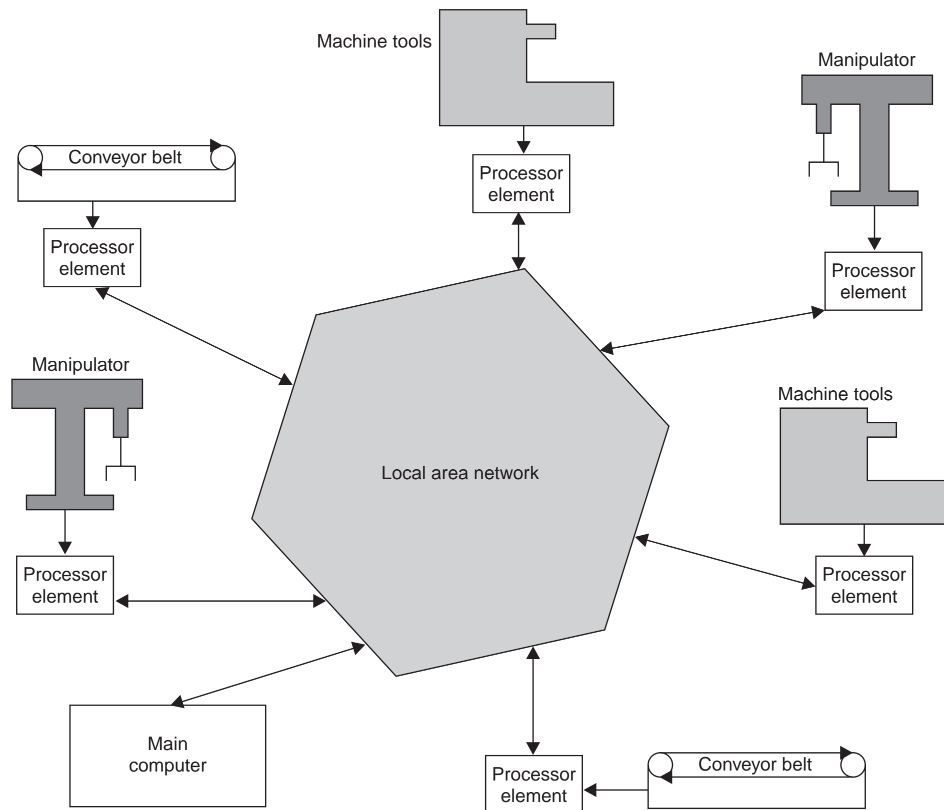

Figure 14.1 A manufacturing distributed embedded system.

where nodes work to no common purpose[1]. The majority of applications one might sensibly embed on multiprocessor architectures – for example command and control, banking (and other transaction-oriented commercial applications), and data acquisition – fall within the definition. A distributed manufacturing-based system is shown in Figure 14.1.

Even modern aircraft designs (both civil and military) have embedded distributed systems. For example, Integrated Modular Avionics (AEEC, 1991) allows more than one processing modules to be interconnected via an ARINC 629 bus, as illustrated in Figure 14.2.

It is useful to classify distributed systems as either **tightly coupled**, meaning that the processing elements, or nodes, have access to a common memory, and **loosely**

[1] However, as communication technology continues to improve, more and more internet-working will fit this definition of a distributed system.

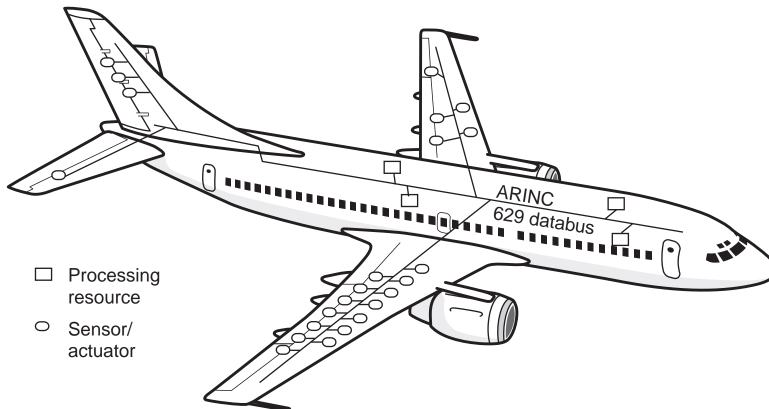

Figure 14.2 A civil avionics distributed embedded system.

coupled, meaning that they do not. The significance of this classification is that synchronization and communication in a tightly coupled system can be effected through techniques based on the use of shared variables, whereas in a loosely coupled system some form of message passing is ultimately required. It is possible for a loosely coupled system to contain nodes which are themselves tightly coupled systems.

This chapter will use the term 'distributed system' to refer to loosely coupled architectures. Also, in general, full connectivity will be assumed between processors – issues associated with the routing of messages and so on will not be considered. For a full discussion on these topics, see Tanenbaum (1998). Furthermore, it will be assumed that each processor will have access to its own clock and that these clocks are loosely synchronized (that is, there is a bound by which they can differ).

A separate classification can be based on the variety of processors in the system. A **homogeneous** system is one in which all processors are of the same type; a **heterogeneous** system contains processors of different types. Heterogeneous systems pose problems of differing representations of program and data; these problems, while significant, are not considered here. This chapter assumes that all processors are homogeneous.

14.2 Overview of issues

So far in this book, the phrase concurrent programming has been used to discuss communication, synchronization and reliability without getting too involved with how processes are implemented. However, some of the issues which arise when distributed applications are considered raise fundamental questions that go beyond mere implementation details. The purpose of this chapter is to consider these issues and their

implications for real-time applications. They are:

- **Language support** – The process of writing a distributed program is made much easier if the language and its programming environment support the partitioning, configuration, allocation and reconfiguration of the distributed application, along with location-independent access to remote resources.
- **Reliability** – The availability of multiple processors enables the application to become tolerant of processor failure – the application should be able to exploit this redundancy. Although the availability of multiple processors enables the application to become tolerant of processor failure, it also introduces the possibility of more faults occurring in the system which would not occur in a centralized single-processor system. These faults are associated with *partial* system failure and the application program must either be shielded from them, or be able to tolerate them.
- **Distributed control algorithms** – The presence of true parallelism in an application, physically distributed processors, and the possibility that processors and communication links may fail, means that many new algorithms are required for resource control. For example, it may be necessary to access files and data which are stored on other machines; furthermore, machine or network failure must not compromise the availability or consistency of those files or data. Also, as there is often no common time reference in a distributed system, each node having its own local notion of time, it is very difficult to obtain a consistent view of the overall system. This can cause problems when trying to provide mutual exclusion over distributed data.
- **Deadline scheduling** – In Chapter 13, the problems of scheduling processes to meet deadlines in a single processor system were discussed. When the processes are distributed, the optimal single processor algorithms are no longer optimal. New algorithms are needed.

These issues are now discussed in turn. However, in one chapter it is difficult to do justice to all the new activities in these areas.

14.3 Language support

The production of a distributed software system to execute on a distributed hardware system involves several steps which are not required when programs are produced for a single processor:

- **Partitioning** is the process of dividing the system into parts (units of distribution) suitable for placement onto the processing elements of the target system.
- **Configuration** takes place when the partitioned parts of the program are associated with particular processing elements in the target system.

- **Allocation** covers the actual process of turning the configured system into a collection of executable modules and downloading these to the processing elements of the target system.
- **Transparent execution** is the execution of the distributed software so that remote resources can be accessed in a manner which is independent of their location.
- **Reconfiguration** is the dynamic change to the location of a software component or resource.

Most languages which have been designed explicitly to address distributed programming will provide linguistic support for at least the partitioning stage of system development. For example, processes, objects, partitions, agents and guardians have all been proposed as units of distribution. All these constructs provide well-defined interfaces which allow them to encapsulate local resources and provide remote access. Some approaches will allow configuration information to be included in the program source, whereas others will provide a separate **configuration** language.

Allocation and reconfiguration, typically, require support from the programming support environment and operating system.

It is, perhaps, in the area of transparent execution where most efforts have been made to achieve a level of standardization across the various approaches. The goal is to make communication between distributed processes as easy and reliable as possible. Unfortunately, in reality, communication often takes places between heterogeneous processors across an unreliable network, and in practice complex communication protocols are required (see Section 14.5). What is needed is to provide mechanisms whereby:

- Processes do not have to deal with the underlying form of messages. For example, they do not need to translate data into bit strings suitable for transmission or to break up the message into packets.
- All messages received by user processes can be assumed to be intact and in good condition. For example, if messages are broken into packets, the run-time system will only deliver them when all packets have arrived at the receiving node and can be properly reassembled. Furthermore, if the bits in a message have been scrambled, the message either is not delivered or is reconstructed before delivery; clearly some redundant information is required for error checking and correction.
- Messages received by a process are the kind that the process expects. The process does not need to perform run-time checks.
- Processes are not restricted to communication only in terms of a predefined built-in set of types. Instead, processes can communicate in terms of values of interest to the application. Ideally, if the application is defined using abstract data types, then values of these types can be communicated in messages.

It is possible to identify three main *de facto* standards by which distributed programs can communicate with each other:

- by using an application programmers interface (API), such as *sockets*, to network transport protocols

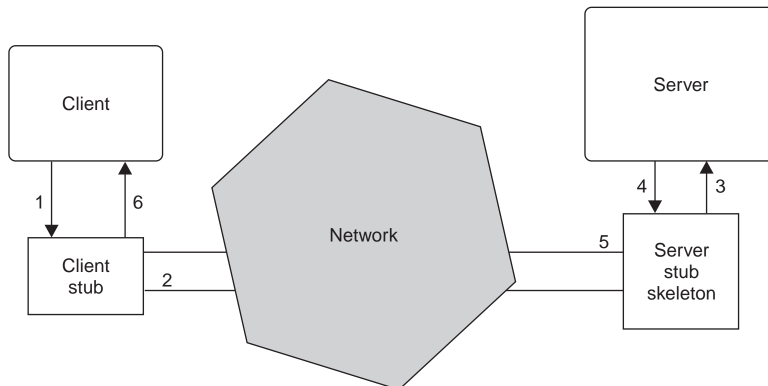

Figure 14.3 The relationship between client and server in an RPC.

- by using the remote procedure call (RPC) paradigm
- by using the distributed object paradigm.

The issue of network protocols will be discussed in Section 14.5 and a Java interface for sockets will be briefly considered in Section 14.4.3. The remainder of this subsection will consider RPC and distributed objects including the Common Object Request Broker Architecture (CORBA).

14.3.1 Remote procedure call

The overriding goal behind the remote procedure call paradigm is to make distributed communication as simply as possible. Typically, RPCs are used for communication between programs written in the same language, for example, Ada or Java. A procedure (server) is identified as being one that can be called remotely. From the server specification, it is possible to generate automatically two further procedures: a **client stub** and a **server stub**. The client stub is used in place of the server at the site on which the remote procedure call originates. The server stub is used on the same site as the server procedure. The purpose of these two procedures is to provide the link between the client and the server in a transparent way (and thereby meet all the requirements laid out in the previous section). Figure 14.3 illustrates the sequence of events in a RPC between a client and a server via the two stub procedures. The stubs are sometimes called **middleware** as they sit between the application and the operating system.

The role of the client stub is to:

- identify the address of the server (stub) procedure;

- convert the parameters of the remote call into a block of bytes suitable for transmission across the network – this activity is often call **parameter marshalling**;
- send the request to execute the procedure to the server (stub);
- wait for the reply from the server (stub) and unmarshal the parameters or any exceptions propagated;
- return control to the client procedure along with the returned parameters, or raise an exception in the client procedure.

The goal of the server stub is to:

- receive requests from client (stub) procedures;
- unmarshal the parameters;
- call the server;
- catch any exceptions that are raised by the server;
- marshal the return parameters (or exceptions) suitable for transmission across the network;
- send the reply to the client (stub).

Where the client and server procedures are written in different languages or are on different machine architectures, the parameter marshalling and unmarshalling mechanisms will convert the data into a machine- and language-independent format (see Section 14.4.4).

14.3.2 The Distributed Object Model

The term **distributed objects** (or **remote objects**) has been used over the last few year in a variety of contexts. In its most general sense, the distributed object model allows:

- the dynamic creation of an object (in any language) on a remote machine;
- the identification of an object to be determined and held on any machine;
- the transparent invocation of a remote method in an object as if it were a local method and irrespective of the language in which the object is written;
- the transparent run-time dispatching of a method call across the network.

Not all systems which support distributed objects provide mechanisms to support all this functionality. As will be shown in the following subsections:

Ada supports the static allocation of objects, allows the identification of remote Ada objects, facilitates the transparent execution of remote methods, and supports distributed run-time dispatching of method calls;

Java allows the code of a Java object to be sent across the network and instances to be created remotely, the remote naming of a Java object, the transparent invocation of its methods, and distributed run-time dispatching;

CORBA allows objects to be created in different languages on different machines, facilitates the transparent execution of remote methods, and supports distributed run-time dispatching of method calls.

14.4 Distributed programming systems and environments

The number of distributed applications is vast, ranging from simple embedded control systems to large complex multi-language generic information processing platforms. It is beyond the scope of this book to discuss fully how these systems can be designed and implemented. However, four approaches will be considered:

(1) occam2 – for simple embedded control systems
(2) Ada – for more complex distributed real-time applications
(3) Java – for single-language Internet-type applications
(4) CORBA – for multi-language multi-platform applications

14.4.1 Occam2

Occam2 has been specifically designed so that programs can be executed in a distributed environment, that of a multi-transputer network. In general, occam2's processes do not share variables so the unit of partitioning is the process itself. Configuration is achieved by the `PLACED PAR` construct. A program constructed as a top-level `PAR`, such as:

```
PAR
  p1
  p2
  p3
  p4
  p5
```

can be distributed, for example as follows:

```
PLACED PAR
  PROCESSOR 1
    p1
  PROCESSOR 2
    PAR
      p2
      p3
  PROCESSOR 3
    PAR
      p4
      p5
```

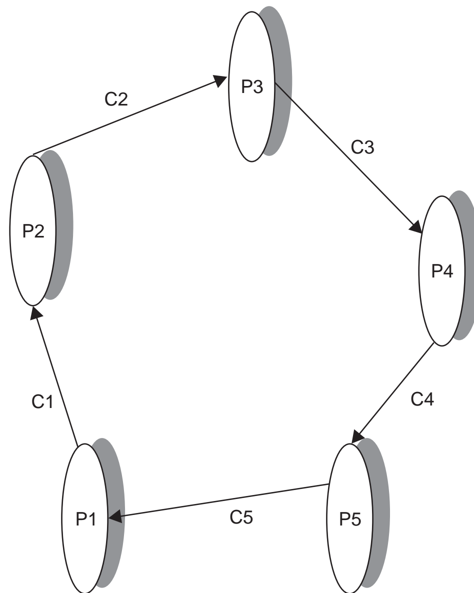

Figure 14.4 Five occam2 processes connected by five channels.

It is important to note that the transformation of the program from one that has a simple `PAR` to one that uses a `PLACED PAR` will not invalidate the program. However, occam2 does allow variables to be read by more than one process on the same processor. Therefore, a transformation may not be possible if the programmer has used this facility.

For the transputers, it is also necessary to associate each external channel with an appropriate transputer link. This is achieved by using the `PLACE AT` construct. For example, consider the above example with the following integer channels shown in Figure 14.4.

The program for execution on a single transputer is:

```
CHAN OF INT c1, c2, c3, c4, c5:
PAR
  p1
  p2
  p3
  p4
  p5
```

If the program is configured to three transputers, as illustrated in Figure 14.5, the occam2 program becomes:

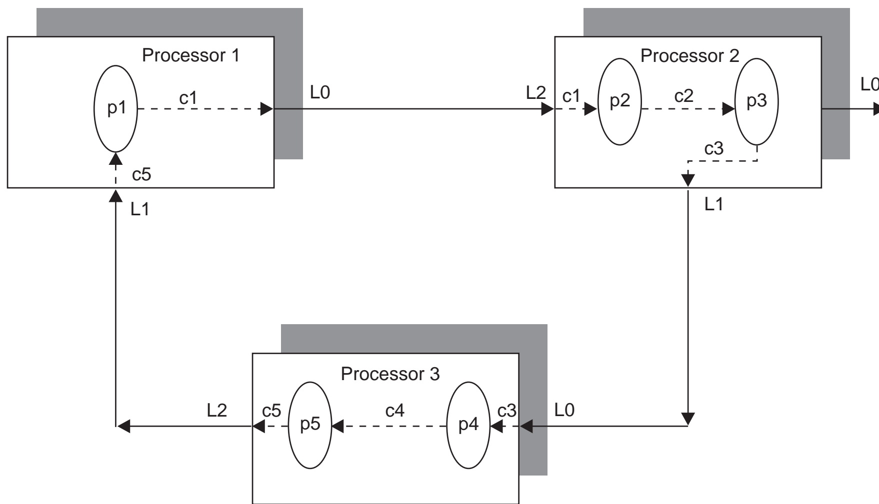

Figure 14.5 Five occam2 processes configured for three transputers.

```
CHAN OF INT c1, c3, c5:
PLACED PAR
  PROCESSOR 1
    PLACE c1 at 0:
    PLACE c5 at 1:
    p1
  PROCESSOR 2
    PLACE c1 at 2:
    PLACE c3 at 1:
    CHAN OF INT c2:
    PAR
      p2
      p3
  PROCESSOR 3
    PLACE c3 at 0:
    PLACE c5 at 2:
    CHAN OF INT c4:
    PAR
      p4
      p5
```

The ease with which occam2 programs can be configured for execution on a distributed system is one of the main attractions of occam2.

Allocation is not defined by the occam2 language nor are there any facilities for reconfiguration. Further, access to resources is not transparent.

Real-time perspective

The occam2 support for real-time is limited. However, within these limitations, the distributed system model is consistent.

14.4.2 Ada 95

Ada defines a distributed system as an

> interconnection of one or more processing nodes (a system resource that has both computational and storage capabilities), and zero or more storage nodes (a system resource that has only storage capabilities, with the storage addressable by more than one processing nodes).

The Ada model for programming distributed systems specifies a **partition** as the unit of distribution. Partitions comprised aggregations of library units (separately compiled library packages or subprograms) that collectively may execute in a distributed target execution environment. The configuration of library units into partitions is not defined by the language; it is assumed that an implementation will provide this, along with facilities for allocation and, if necessary, reconfiguration.

Each partition resides at a single execution site where all its library units occupy the same logical address space. More than one partition may, however, reside on the same execution site. Figure 14.6 illustrates one possible structure of a partition. The arrows represent the dependencies between library units. The principal interface between partitions consists of one or more package specifications (each labelled 'partition interface library unit' in Figure 14.6).

Partitions may be either **active** or **passive**. The library units comprising an active partition reside and execute upon the same processing element. In contrast, library units comprising a passive partition reside at a storage element that is directly accessible to the nodes of different active partitions that reference them. This model ensures that active partitions cannot directly access variables in other active partitions. Variables can only be shared directly between active partitions by encapsulating them in a passive partition. Communication between active partitions is defined in the language to be via remote subprogram calls (however, an implementation may provide other communication mechanisms).

Categorization pragmas

To aid the construction of distributed programs, Ada distinguishes between different categories of library units, and imposes restrictions on these categories to maintain type consistency across the distributed program. The categories (some of these are useful in their own right, irrespective of whether the program is to be distributed) are designated by the following pragmas:

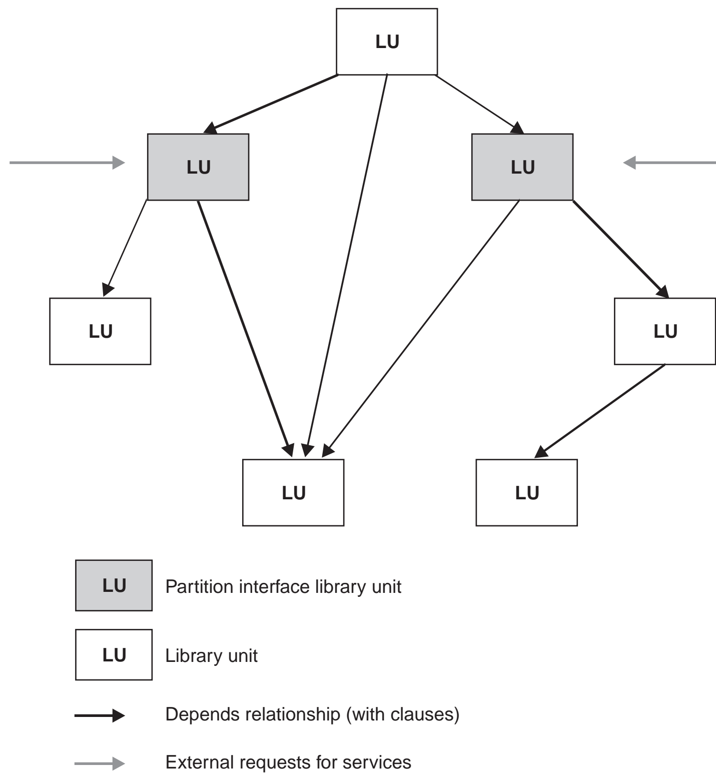

Figure 14.6 The structure of a partition.

`Preelaborate`

A preelaborable library unit is one that can be elaborated without execution of code at run-time.

`Pure`

Pure packages are preelaboratable packages with further restrictions which enable them to be freely replicated in different active or passive partitions without introducing any type inconsistencies. These restrictions concern the declaration of objects and types; in particular, variables and named access types are not allowed unless they are within a subprogram, task unit or protected unit.

`Remote_Types`

A `Remote_Types` package is a preelaboratable package that must not contain any variable declarations within the visible part.

`Shared_Passive`

`Shared_Passive` library units are used for managing global data shared between active partitions. They are, therefore, configured on storage nodes in the distributed system.

`Remote_Call_Interface`

A `Remote_Call_Interface` package defines the interface between active partitions. Its body exists only within a single partition. All other occurrences will have library stubs allocated.

The specification of a `Remote_Call_Interface` package must be preelaborable; in addition other restrictions apply, for example it must not contain the definition of a variable (to ensure no remote data access).

A package which is not categorized by any categorization pragma is called a *normal* library package. If it is included in more than one partition, then it is replicated and all types and objects are viewed as distinct. For example, the `Calendar` package is, in this regard, normal.

The above pragmas facilitate the distribution of an Ada program and ensure that illegal partitionings (which allow direct remote variable access between partitions) are easily identifiable.

Remote communication

The only predefined way in which active partitions can communicate directly is via remote subprogram calls. They can also communicate indirectly via data structures in passive partitions.

There are three different ways in which a calling partition can issue a remote subprogram call:

- by calling a subprogram which has been declared in a remote call interface package of another partition directly;
- by dereferencing a pointer to a remote subprogram;
- by using run-time dispatching to a method of a remote object.

It is important to note that, in the first type of communication, the calling and the called partitions are statically bound at compile time. However, in the latter two, the partitions are dynamically bound at run-time. Hence Ada can support transparent access to resources.

Program 14.1 The Ada `System.RPC` package.

```
with Ada.Streams;
package System.RPC is

  type Partition_ID is range 0 ..
implementation_defined;

  Communication_Error : exception;

  type Params_Stream_Type ...

  -- Synchronous call
  procedure Do_RPC(
          Partition  : in Partition_ID;
          Params     : access Params_Stream_Type;
          Result     : access Params_Stream_Type);

  -- Asynchronous call
  procedure Do_APC(
          Partition  : in Partition_ID;
          Params     : access Params_Stream_Type);

  -- The handler for incoming RPCs
  type RPC_Receiver is access procedure(
          Params     : access Params_Stream_Type;
          Result     : access Params_Stream_Type);

  procedure Establish_RPC_Receiver(Partition : Partition_ID;
          Receiver : in RPC_Receiver);

private
  ...
end System.RPC;
```

Many remote calls contain only ‘in’ or ‘access’ parameters (that is, data that is being passed in the same direction as the call) and a caller may wish to continue its execution as soon as possible. In these situations it is sometimes appropriate to designate the call as an *asynchronous* call. Whether a procedure is to be called synchronously or asynchronously is considered by Ada to be a property of the procedure and not of the call. This is indicated by using a pragma `Asynchronous` when the procedure is declared.

Ada has defined how distributed programs can be partitioned and what forms of remote communication must be supported. However, the language designers were keen not to overspecify the language and not to prescribe a distributed run-time support system for Ada programs. They wanted to allow implementors to provide their own network communication protocols and, where appropriate, allow other ISO standards

to be used; for example the ISO Remote Procedure Call standard. To achieve these aims, the Ada language assumes the existence of a standard implementation-provided subsystem for handling all remote communication (the Partition Communication Subsystem, PCS). This allows compilers to generate calls to a standard interface without being concerned with the underlying implementation.

The package defined in Program 14.1 illustrates the interface to the remote procedure (subprogram) call (RPC) support system which is part of the PCS.

The type `Partition_Id` is used to identify partitions. For any library-level declaration, `D`, `D'Partition_Id` yields the identifier of the partition in which the declaration was elaborated. The exception `Communication_Error` is raised when an error is detected by `System.RPC` during a remote procedure call. An object of stream type `Params_Stream_Type` is used for marshalling (translating data into an appropriate stream-oriented form) and unmarshalling the parameters or results of a remote subprogram call, for the purposes of sending them between partitions. The object is also used to identify the particular subprogram in the called partition.

The procedure `Do_RPC` is invoked by the calling stub after the parameters are flattened into the message. After sending the message to the remote partition, it suspends the calling task until a reply arrives. The procedure `Do_APC` acts like `Do_RPC` except that it returns immediately after sending the message to the remote partition. It is called whenever the `Asynchronous` pragma is specified for the remotely called procedure. `Establish_RPC_Receiver` is called immediately after elaborating an active partition, but prior to invoking the main subprogram, if any. The `Receiver` parameter designates an implementation-provided procedure that receives a message and calls the appropriate remote call interface package and subprogram.

Real-time perspective

Although Ada defines a coherent real-time model for single and multiprocessor systems, it has very limited support for *distributed real-time* systems. There is no integration between the Distributed Systems Annex and the Real-Time Annex. Arguably, the technology of language support in this area is not sufficiently widely accepted to merit standardization.

14.4.3 Java

There are essentially two ways in which to construct distributed Java applications:

(1) execute Java programs on separate machines and use the Java networking facilities;

(2) use remote objects.

Java networking

In Section 14.5, two network communication protocols will be introduced: UDP and TCP. These are the prominent communication protocols in use today and the Java environment provide classes (in the `java.net` package) which allow easy access to them. The API to these protocols is via the `Socket` class (for the reliable TCP protocol) and the `DatagramSocket` class (for the UDP protocol). It is beyond the scope of this book to consider this approach in detail – see the Further Reading section at the end of this chapter for alternative sources of information.

Remote objects

Although Java provides a convenient way of accessing network protocols, these protocols are still complex and are a deterrent to writing distributed applications. Consequently, Java supports the distributed object communication model through the notion of **remote objects**.

The Java model is centred on the use of the `java.rmi` package which builds on top of the TCP protocol. In this package is the `Remote` interface:

```
public interface Remote { };
```

This is the starting point for writing distributed Java applications. Extensions of this interface are written to provide the link between the clients and servers. For example, consider a server which wishes to return details of the local weather forecast for its location. An appropriate interface might be:

```
public interface WeatherForecast extends java.rmi.Remote
  // shared between clients and server
{
  public Forecast getForecast() throws RemoteException;
}
```

The method `getForecast` must have a `RemoteException` class in its throws list so that the underlying implementation can indicate that the remote call has failed.

`Forecast` is an object which has details of today's forecast. As the object will be copied across the network, it must implement the `Serializable` interface[2]

```
public class Forecast implements java.io.Serializable
{
  public String Today() {
    String today = "Wet";

    return today ;
  }
}
```

[2]Like the `Remote` interface, the `Serializable` interface is empty. In both cases it acts as a tag to give information to the compiler. For the `Serializable` interface, it indicates that the object can be converted into a stream of bytes suitable for I/O.

Once the appropriate remote interface has been defined, a server class can be declared. Again, it is usual to indicate that objects of the class can potentially be called remotely by extending one of the predefined classes in the package `java.rmi.server`. Currently, there are two classes: `RemoteServer` (which is an abstract class derived from the `Remote` class) and `UnicastRemoteObject` which is a concrete extension of `RemoteServer`. The latter provides the class for servers which are non-replicated and have a point-to-point connection to each client using the TCP protocol. It is anticipated that future extension to Java might provide other classes such as a replicated server using a multicast communication protocol.

The following examples shows a server class which provides the weather forecast for the county of Yorkshire in the UK.

```
public class YorkshireWeatherForecast extends UnicastRemoteObject
                                      implements WeatherForecast
{

  public YorkshireWeatherForecast() throws RemoteException
  {
    super(); // call parent constructor
  }

  public Forecast getForecast() throws RemoteException
  {
    ...
  }
}
```

Once the server class has been written, it is necessary to generate the server and client stubs for each of the methods that can be called remotely. Note that Java uses the term **skeleton** for the server stub. The Java programming environment provides a tool called 'rmic' which will take a server class and automatically generate the appropriate client stub and server skeleton.

All that is now required is for the client to be able to acquire a client stub which accesses the server object. This is achieved via a registry. The registry is a separate Java program which executes on each host machine which has server objects. It listens on a standard TCP port and provides an object of the class `Naming` extracted in Program 14.2.

Each server object can use the `Naming` class to bind its remote object to a name. Clients then can access a remote registry to acquire a reference to the remote object. Of course, this is a reference to a client stub object for the server. The client stub is loaded into the client's machine. Once the reference has been obtained, the server's methods can be invoked.

Real-time perspective

The facilities described above are those of Standard Java which is not intended to operate within real-time constraints. Real-Time Java is currently silent on the issue of distributed

Program 14.2 An extract of the Java Naming class.

```
public final class Naming
{
  public static void bind(String name, Remote obj)
    throws AlreadyBoundException, java.net.MalformedURLException,
           UnknownHostException, RemoteException;
  // bind the name to the obj
  // name takes the form of a URL such as
  //           rmi://remoteHost:pot/objectName

  public static Remote lookup (String name)
    throws NotBoundException, java.net.MalformedURLException,
           UnknownHostException, RemoteException;
  // looks up the name in the registry and returns a remote object

 ...
}
```

real-time programming. However, this is a topic which will be addressed within the next few years.

14.4.4 CORBA

The Common Object Request Broker Architecture (CORBA) provides the most general distributed object model. Its goal is to facilitate interoperability between applications written in different languages, for different platforms supported by middleware software from different vendors. It was designed by the Object Management Group (OMG) – a consortium of software vendors, software developers and end users – according to the Object Management Architectural Model, which is depicted in Figure 14.7.

At the heart of the architecture is the **Object Request Broker** (ORB). This is a software communication bus that provides the main infrastructure that facilitates interoperability between heterogeneous applications. The term CORBA often refers to the ORB. The other components of the architecture are:

Object Services – a collection of basic services which support the ORB; for example, support for object creation, naming and access control, and tracking relocated objects.

Common Facilities – a set of functions which is common across a wide range of application domains; for example, user interfaces, document and database management.

Domain Interfaces – a group of interfaces which support particular application domains such as banking and finance or telecommunications.

Application Interfaces – the end users' specific application interfaces.

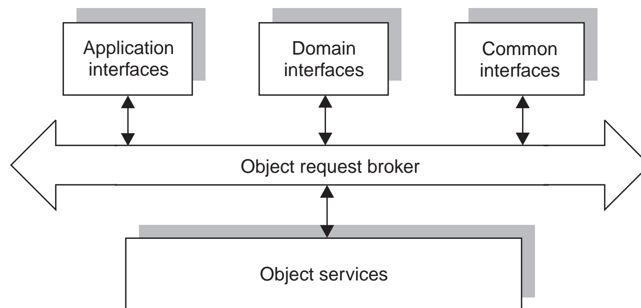

Figure 14.7 The Object Management Architecture Model.

To ensure interoperability between ORBs from different vendors, CORBA defines a General Inter-ORB Protocol which sits on top of TCP/IP (see section 14.5.2).

Central to writing CORBA applications is the **Interface Definition Language** (IDL) . A CORBA interface is similar in concept to the Java remote interface discussed in the previous section. The IDL (which is like the C++ language) is used to describe the facilities to be provided by an application object, the parameters to be passed with a given method and its return values, along with any object attributes. An example interface for the weather forecast application given in the previous section is shown below.

```
interface WeatherForecast {
  void GetForecast(out Forecast today);
}
```

Once the IDL for the application is defined, tools are used to 'compile' it. The IDL compiler generates several new files in one of a number of existing programming languages such as Java or Ada. The files include:

- client stubs that provide a communication channel between the client and the ORB; and
- a server skeleton which enables the ORB to call functions on the server.

The code for the server and clients can now be written. The server consists of two parts: the code for the application object itself and the code for the server process. The application object has to be associated with the server skeleton. The way this is achieved is dependent on the target language. For example in Java, it can be done by producing a class which is a subclass of the generated server skeleton. The methods for the application objects are then completed. The server process itself can be written as a main program which creates the application object and initializes the ORB and tells it that the object is ready to receive client requests. The structure of the client is similar.

In reality, CORBA provides many more facilities than those described above. As well as the static association between client and server, CORBA clients can dynamically discover a server's IDL interface without prior knowledge of the server details. Furthermore, the services that support the ORB allow for a wide range of functionality such as an event service, transaction and concurrency control, persistent objects and trading services. Applications access these services via the Portable Object Adapter (POA). This is a library which provides the run-time environment for a server object.

Real-time perspective

Although, in general, CORBA provides comprehensive support for distributed objects, from this book's perspective, one of the main limitation of CORBA has been its lack of support for real-time applications. The ORBs were not designed to operate within time constraints and the result is that many are inappropriate for soft real-time systems, let alone systems which have to operate within stringent timing constraints. Furthermore, the basic communication models provided by CORBA are: a synchronous RPC where the client must wait for the server to reply, and a deferred synchronous RPC where the client thread continues and subsequently polls for the reply. As was illustrated in Chapter 13, synchronous communication models are more difficult to analyze for their timing properties than asynchronous models and the results are more pessimistic.

For these reasons and the increasing use of CORBA in real-time applications, the OMG have set out over the last few years to address the performance problems associated with the main CORBA standard. They have tackled the problems from three perspectives:

(1) Minimum CORBA

(2) CORBA Messaging

(3) Real-Time CORBA

Minimum CORBA is a subset of the CORBA 2.2 specification which omits many services including all those associated with dynamic configurations between client and server objects. The subset is targeted at embedded systems with limited resources.

CORBA Messaging is an extension to the CORBA standard to support asynchronous message passing and quality of service parameters.

The Real-Time CORBA (RT CORBA) (Object Management Group, 1999) specification defines mechanisms that support predictability of distributed CORBA applications. It assumes that real-time behaviour is obtained by using fixed-priority based scheduling and is, therefore, compatible with most real-time operating systems – especially those supporting the Real-Time POSIX interfaces. The key aspect of RT CORBA is that it allows applications to configure and control both processor and communication resources. These facilities will now be briefly described.

Managing processor resources

The RT CORBA specification allows client and server applications to manage the following properties.

- The priority at which servers process client requests. RT CORBA specifies global CORBA priorities and mechanism to map these priorities onto the priority range of the particular real-time operating system hosting the RT ORB. Using these mapping, three global priority models are supported: (1) a server declared model in which the server specifies the priority of the requests it services, (2) a client propagated model in which the client specifies the priority and this is propagated to the server and (3) a priority transformation model in which the priority of a request is based on external factors such as current server load or the state of the global scheduling service.
- The degree of multithreading in a server. RT CORBA controls the degree of multithreading by the concept of thread pools. Using a Real-Time POA, server applications can specify: the default number of threads that are initially created, the maximum number of threads that can be created dynamically, and the default priority of all the threads. These threads are allocated from a thread pool which may or may not be shared between applications. Further flexibility is given by the concept of thread pools with lanes. Here, not only can the overall amount of concurrency be controlled, but also the amount of work performed at a particular priority.
- The impact of priority inheritance and priority ceiling protocols. RT CORBA defines POSIX-like mutexes to ensure consistency of synchronization protocols encapsulating shared resources.

Managing network resources

Although having transparent access to resources eases the writing of general-purpose distributed applications, it makes it impossible to do any realistic analysis of real-time performance. For this reason, RT CORBA allows explicit connections between client and server to be set up, for example, at system configuration time. It is also possible to control how client requests are sent over these connections. RT CORBA facilitates multiple connections between a client and a server in order to reduce priority inversion problems due to the use of non real-time transport protocols. Furthermore, private transport connections are also supported – so that a call from a client to a server can be made without any fear of the call having to compete with other calls which have been multiplexed on the same connection.

Even with the above support, it is still advantageous to be able to set any quality of service parameters of the specific communication protocols underlying the inter-ORB communication protocol. RT CORBA provides interfaces to select and configure these properties from both the client and the server side.

The scheduling service

From the above discussion it should be clear that RT CORBA provides many mechanisms to facilitate real-time distributed CORBA applications. However, setting up all the required parameters (so that a consistent scheduling policy is obtained) can be difficult. For this reason, RT CORBA allows an application to specify its scheduling requirements in terms of characteristics such as its period, its worst-case execution time, its criticality etc. This is done offline and each application scheduled entity (called an *activity*) is allocated a textual name. At run-time, interfaces are provided which allow the application to schedule a named activity via a *scheduling service*. This service sets all the necessary priority parameters to implement a specific scheduling policy such as deadline or rate monotonic scheduling.

14.5 Reliability

It seems almost paradoxical that distribution can provide the means by which systems can be made more reliable yet at the same time introduce more potential failures in the system. Although the availability of multiple processors enables the application to become tolerant of processor failure, it also introduces the possibility of faults occurring in the system which would not occur in a centralized single processor system. In particular, multiple processors introduce the concept of a partial system failure. In a single processor system, if the processor or memory fails then normally the whole system fails (sometimes the processor may be able to continue and recover from a partial memory failure, but in general the system will crash). However, in a distributed system, it is possible for a single processor to fail while others continue to operate. In addition, the propagation delay through the underlying communications network is variable and messages may take various routes. This, in conjunction with an unreliable transmission medium, may result in messages being lost, corrupted, or delivered in an order different to the order in which they were sent. The increased complexity of the software necessary to tolerate such failures can also threaten the reliability of the system.

14.5.1 Open systems interconnections

Much effort has been expended on communication protocols for networks and distributed systems. It is beyond the scope of this book to cover this in detail; rather the reader should refer to the Further Reading section at the end of this chapter. In general, communication protocols are layered to facilitate their design and implementation. However, many different networks exist, each with its own concept of a 'network architecture' and associated communication protocol. Consequently, without some international standards it is extremely difficult to contemplate interconnecting systems of different origins. Standards have been defined by the International Organization for Standardization (ISO) and involve the concept of *Open Systems Interconnections*, or *OSI*. The term 'open' is used to indicate that, by conforming to these standards, a system will be open to all other systems in the world that also obey the same standards.

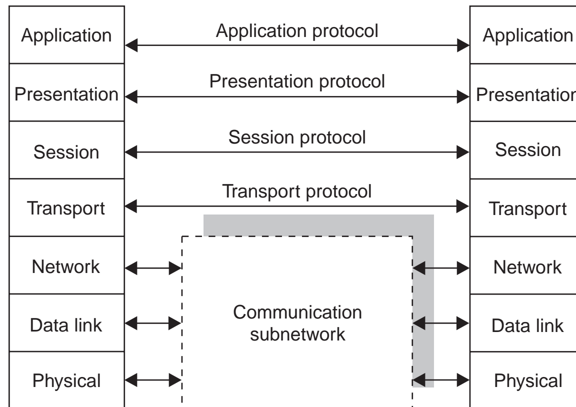

Figure 14.8 The OSI reference model.

These standards have become known as the *OSI Reference Model.* It should be stressed that this model is *not* concerned with specific applications of computer communication networks but with the *structuring* of the communication protocols required to provide a reliable, manufacturer independent communication service. The OSI Reference Model is a layered model. The layers are shown in Figure 14.8.

The basic idea of layering is that each layer adds to the services provided by the lower layers in order to present a service to higher layers. Viewed from above a particular layer, the ones below it may be considered as a black box which implements a service. The means by which one layer makes use of the service provided by the lower layers is through that layer's interface. The interface defines the rules and format for exchanging information across the boundary between adjacent layers. The modules which implement a layer are usually known as **entities**.

In networks and distributed systems, each layer may be distributed across more than one machine; in order to provide its service, entities in the same layer on different machines may need to exchange information. Such entities are known as **peer entities**. A **protocol** is the set of rules which governs communication between peer entities.

The OSI model itself does not define protocol standards; by breaking up the network's function into layers, it does suggest where protocol standards should be developed but these standards are outside the model itself. Such standards, however, have been developed.

The functions of each layer are now briefly described.

(1) **The Physical Layer**

The physical layer is concerned with transmitting raw data over a communication channel. Its job is to make sure that, in the absence of errors, when one side sends a 1 bit it is received as a 1 bit and not a 0 bit.

(2) **The Data Link Layer**

The data link layer converts a potentially unreliable transmission channel into a reliable one for use by the network layer. It is also responsible for resolving any contention for access to the transmission channel between nodes connected to the channel.

In order for the data link layer to provide a reliable communication channel, it must be able to correct errors. There are two basic and familiar techniques used: *forward error control* and *backward error control.* Forward error control requires enough redundant information in each message to correct any errors which may occur in its transmission. In general, the amount of redundancy required increases rapidly as the number of information bits increases. Backward error control requires only that the error be detected; once detected, a retransmission scheme can be employed to obtain the correct message (this is the job of the data link layer). Backward error control predominates in the world of networks and distributed systems.

Most backward error control techniques incorporate the notion of a calculated *checksum* which is sent with the message and describes the content of the message. On receipt, the checksum is recalculated and compared with the one sent. Any disagreement indicates that a transmission error has occurred. At this point, there are three basic classes of service that the Data Link Layer can offer: an unacknowledged connectionless service, an acknowledged connectionless service or a connection-oriented service. With the first, no further service is provided. The sender is unaware that the message has not been received intact. With the second, the sender is informed every time a message is received correctly; the absence of this acknowledgement message within a certain time period indicates that an error has occurred. The third service type establishes a connection between the send and the receiver and guarantees that all messages are received correctly and are received in order.

(3) **The Network Layer**

The network layer (or communication subnet layer) is concerned with how information from the transport layer is routed through the communication subnet to its destinations. Messages are broken down into packets which may be routed via different paths; the network layer must reassemble the packets and handle any congestion that may occur. There is no clear agreement as to whether the network layer should attempt to provide a perfect communication channel through the network. Two extremes in the services provided can be identified: *virtual circuits* (connection-oriented) and *datagrams* (connectionless). With virtual circuits, a perfect communication channel is provided. All messages packets arrive and do so in sequence. With a datagram service, the network layer attempts to deliver each packet in isolation from the others. Consequently, messages may arrive out of order, or may not arrive at all.

The physical, data link and network layers are network-dependent and their detailed operation may vary from one type of network to another.

(4) **The Transport Layer**

The transport layer (or host-to-host layer) provides reliable host-to-host commu-

nication for use by the session layer. It must hide all details of the communication subnet from the session layer in order that one subnet can be replaced by another. In effect, the transport layer shields the customer's portion of the network (layers 5–7) from the carrier's portion (layers 1–3).

(5) **The Session Layer**
The role of the session layer is to provide a communication path between two application-level processes using the facilities of the transport layer. The connection between users is usually called a session and may include a remote login or a file transfer. The operations involved in setting up a session (called binding) include authentication and accounting. Once the session has been initiated, the layer must control data exchange, and synchronize data operations between the two processes.

(6) **The Presentation Layer**
The presentation layer performs generally useful transformations on the data to overcome heterogeneous issues with respect to the presentation of data to the applications. For example, it allows an interactive program to converse with any one of a set of incompatible display terminals. It may undertake text compression or encryption.

(7) **The Application Layer**
The application layer provides the high-level functions of the network, such as access to databases, mail systems and so on. The choice of the application may dictate the level of services provided by the lower layers. Consequently, particular application areas may specify a set of protocols throughout all seven layers which are required to support the intended distributed processing function. For example, an initiative by General Motors has defined a set of protocols to achieve open interconnection within an automated manufacturing plant. These are called *manufacturing automation protocols* (MAP).

14.5.2 TCP/IP layers

The OSI model is now somewhat outdated and does not directly address the issue of Internet working. The TCP/IP Reference Model has only five layers and is shown in Figure 14.9.

The physical layer is equivalent to the ISO physical layer. The network interface layer performs the same functions as the ISO data link Layer. The Internet layer specifies the format of the packets to be sent across the internet and performs all the routing functions associated with the ISO network layer. It is here that the Internet Protocol (IP) is defined.

The transport layer is equivalent to the ISO transport layer; it provides two protocols: the User Data Protocol (UDP) and the Transmission Control Protocol (TCP). UDP provides an unreliable connectionless protocol which allows efficient access to the IP protocol in the layer below. The TCP protocol provides a reliable end-to-end byte stream protocol.

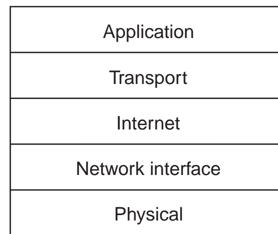

Figure 14.9 The TCP/IP Reference Model.

14.5.3 Lightweight protocols and local area networks

The OSI and TCP/IP model were developed primarily for wide area networks to enable open access; wide area networks were characterized by low bandwidth communication with high error rates. Most distributed embedded systems will use local area network technology and will be closed to the outside world. Local area networks are characterized by high bandwidth communication with low error rates. They may use a range of communication technologies, such as broadcast technology (for example, as does the Ethernet) or point-to-point switch-based technology (for example, ATM (Asynchronous Transfer Mode)). Consequently, although it is possible to implement language-level interprocess communication using the OSI or TCP/IP approach (for example, Java RMI), in practice the expense is often prohibitive. Thus many designers tailor the communication protocols to the requirements of the language (the application) and the communication medium. These are called *lightweight* protocols. A key issue in their design is the degree with which they tolerate communication failures.

At first glance, it appears that completely reliable communication is essential if efficient, reliable distributed applications are to be written. However, this may not always be the case. Consider the types of error that can be introduced by a distributed application. If two distributed processes are communicating and synchronizing their activities to provide a service, potential errors can occur from:

- transient errors resulting from interference on the physical communication medium
- design errors in the software responsible for masking out transient errors in the communication subsystems
- design errors in the protocols between the server processes and any other servers needed in the provision of the service
- design errors in the protocol between the two server processes themselves.

To protect against the latter error, it is necessary for the server processes to provide application-level checks (end-to-end checks). The **end-to-end** argument of system design (Saltzer et al., 1984) states that given the necessity of such checks for provision

of a reliable service, it is not necessary to repeat these checks at lower levels in the protocol hierarchy, particularly when the communication medium (for example, local area networks such as an Ethernet or Token Ring) provides a low error rate (but not perfect) transmission facility. In these cases, it may be better to have a fast, less than 100% reliable, communication facility than a slower 100% reliable facility. Applications which require high reliability can trade off efficiency for reliability at the application level. However, applications that require fast (but not necessarily reliable) service *cannot* trade off reliability for efficiency if the other approach is taken.

There are standards for local area network communication protocols, particularly at the data link layer, which is divided into two sub-layers: Medium Access Control (MAC) and Logical Link Control (LLC). MAC is concerned with the interface to the physical communication media, and standards exists for CSMA/CD (Carrier Sense Multiple Access with Collision Detection) buses (for example, Ethernet), packet switching (for example, ATM), token buses and token rings. The LLC layer is concerned with providing a connectionless or connection-oriented protocol.

As described earlier, a common language-oriented lightweight protocol is the remote procedure call (ISO/IEC JTC1/SC21/WG8, 1992). This is normally implemented directly on top of a basic communication facility provided by the local area network (for example, the LLC layer). With languages like Java and Ada, remote procedure calls, in the absence of machine failures, are considered to be reliable. That is, for each remote procedure call, if the call returns then the procedure has been executed once and once only; this is often called **exactly once** RPC semantics. However, in the presence of machine failure, this is difficult to achieve because a procedure may be partially or totally executed several times depending on where the crash occurred and whether the program is restarted. Ada assumes that the call is executed *at most once* because there is no notion of restarting part of an Ada program following failure.

For a real-time local area network and its associated protocols, it is important to provide bounded and known delays in message transmissions. This topic will be returned to in Section 14.7.2.

14.5.4 Group communication protocols

The remote procedure call (or remote method call) is a common form of communication between clients and servers in a distributed systems. However, it does restrict communication to be between two processes. Often, when groups of processes are interacting (for example, performing an atomic action), it is necessary for communication to be sent to the whole group. A **multicast** communication paradigm provides such a facility. Some networks, for example Ethernet, provide a hardware multicast mechanism as part of their data link layer. If this is not the case, then further software protocols must be added.

All networks and processors are, to a greater or lesser extent, unreliable. It is therefore possible to design a family of group communication protocols, each of which provides a multicast communication facility with specific guarantees:

- **unreliable multicast** – no guarantees of delivery to the group is provided; the multicast protocol provides the equivalent of a datagram-level of service;
- **reliable multicast** – the protocol makes a best-effort attempt to deliver the message to the group, but offers no absolute guarantee of delivery;
- **atomic multicast** – the protocol guarantees that if one process in the group receives the message then all members of the group receive the message; hence the message is delivered to all of the group or none of them;
- **ordered atomic multicast** – as well as guaranteeing the atomicity of the multicast, the protocol also guarantees that all members of the group will receive messages from different senders in the same order.

The more guarantees the protocols give, the greater the cost of their implementation. Furthermore, the cost will also vary depending on the failure model used (see Section 5.2).

For atomic and ordered atomic multicasts, it is important to be able to bound the time taken by the implementation algorithms to terminate. Without this, it is impossible to predict their performance in a hard real-time system.

Consider a simple ordered atomic multicast which has the following failure model:

- processors fail silently
- all communication failures are fail omission
- no more than N consecutive network omission failures occur
- the network is fully connected and there is no network partitioning.

To achieve atomic message transmission, all that is required is to transmit each message $N + 1$ times; this is called message **diffusion**. To achieve the ordered property using diffusion requires that the time required to complete each message transmission be known. Assume that the worst-case transmission value for all messages is T_D and that clocks in the network are loosely synchronized with a maximum difference of C_Δ. Each message is time-stamped by its sender with the value of its local clock (C_{sender}). Each recipient can deliver the message to its process when its local clock is greater than $C_{sender} + T_D + C_\Delta$, as it is by this time that all recipients are guaranteed to have received the message and the message becomes **valid**. These messages can be ordered according to their validity times. If two messages have identical validity times an arbitrary order can be imposed (such as using the network address of the processor). Figure 14.10 illustrates the approach for $N = 1$.

Note that although the above simple algorithm guarantees that all processors receive and process the messages in the same order, it does not guarantee that the order is the actual order the messages were sent. This is because the value used to determine the order is based on the local clocks of the processors which can differ by C_Δ.

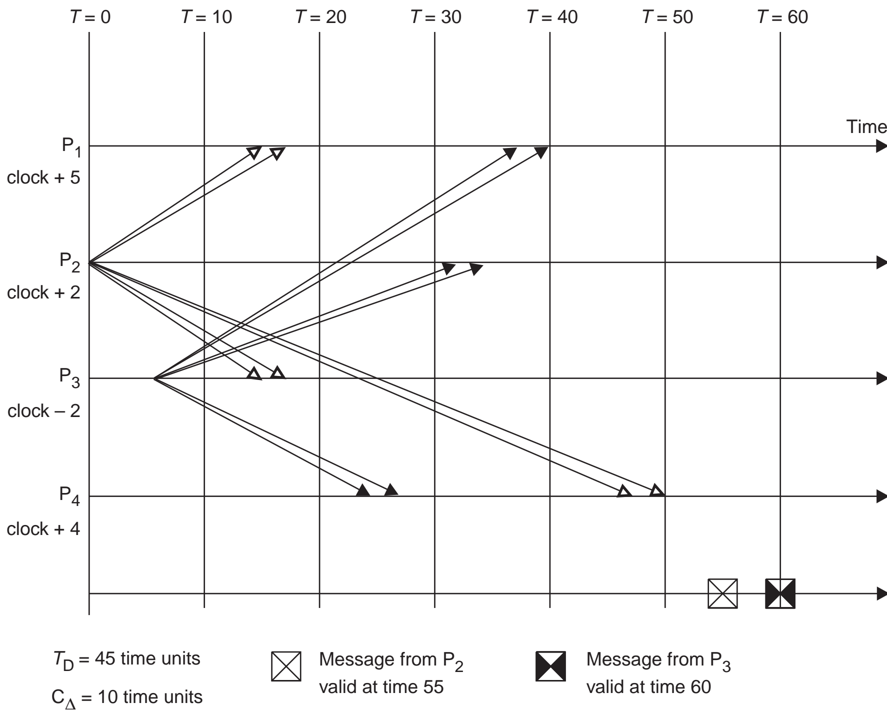

Figure 14.10 A simple ordered atomic multicast based on diffusion.

14.5.5 Processor failure

In Chapter 5, two general approaches for providing fault-tolerant hardware and software were identified; those with static (masking) and those with dynamic redundancy. Although hardware fault tolerance does have a major role to play in achieving reliability in the presence of processor and communication failure, an excessive amount of hardware is required to implement triple modular redundancy, and consequently it is expensive (but fast). This section, therefore, concentrates on the provision of fault tolerance through the use of software methods.

For the time being, it will be assumed that all processors in the system are **fail-silent**. This means that if a processor malfunctions in any way then it will suffer a permanent omission failure. If processors are not fail-silent then it is possible that they will send invalid messages to each other. This would introduce an extra level of complexity into the following discussion.

Tolerating processor failure through static redundancy

N-version programming was discussed in Chapter 5 in the context of achieving static tolerance to design faults. Clearly, if each of the versions in an *N*-version program resides on a different processor then this approach will also provide tolerance to processor failure. However, even if no design diversity is employed, it may still be desirable to replicate identical copies of some system components to obtain the required availability. This is often called **active replication**.

Suppose that an application is designed according to the distributed object model. It is possible to replicate objects on different processors and even vary the degree of replication according to the importance of the particular object. For objects to be replicated transparently to the application programmer, they must have deterministic behaviour. This means that for a given sequence of requests to an object the behaviour of the object is predictable. If this was not the case, the states of each of the replicas in a replicated object set could end up being different. Consequently, any request made to that object set could produce a range of results. A set of objects must, therefore, be kept consistent. Its members must not diverge.

If objects are to be replicated, it is also necessary to replicate each remote method invocation. Furthermore, it is necessary to have exactly once RPC semantics. As is illustrated in Figure 14.11, each client object will potentially execute a one-to-many procedure call and each server procedure will receive a many-to-one request. The run-time system is responsible for coordinating these calls and ensuring the required semantics. This entails periodically probing server sites with outstanding calls to determine their status; a failure to respond will indicate a processor crash. In effect, the run-time system must support some form of membership protocol and an ordered atomic multicast group communication protocol.

An example of a language which explicitly allows replication (at the process level) is Fault-Tolerant Concurrent C (Cmelik et al., 1988). The language assumes that processors have a fail-stop failure model and provides a distributed consensus protocol to ensure that all replicas behave in a deterministic manner. Fault-Tolerant Concurrent C has a communication and synchronization model that is very similar to Ada, and therefore it has to ensure that if a particular branch of a select statement is taken in one replica then the same branch is taken in all replicas (even though the replicas are not executing in lock-step). There have also been attempts to make Ada fault-tolerant, for example, (Wellings and Burns, 1996; Wolf, 1998).

Although transparent replication of objects is an attractive approach to providing tolerance of processor failure, the cost of the underlying multicast and agreement protocols may be prohibitive for a hard real-time system. In particular, where objects are active and contain more than one task (with potentially nested tasks), it is necessary for the run-time agreement protocol (which ensures consistency between replicas) to be executed at every scheduling decision. Providing **warm** standbys with periodic state saving may offer a cheaper solution. With such an approach, an object would be replicated on more than one processor. However, unlike full replication, only one copy of the object is active at any one time. The state of this object is periodically saved (typically before and after communication with another object) on the processors with residing replicas. If the primary node fails a standby can be started from the last checkpoint.

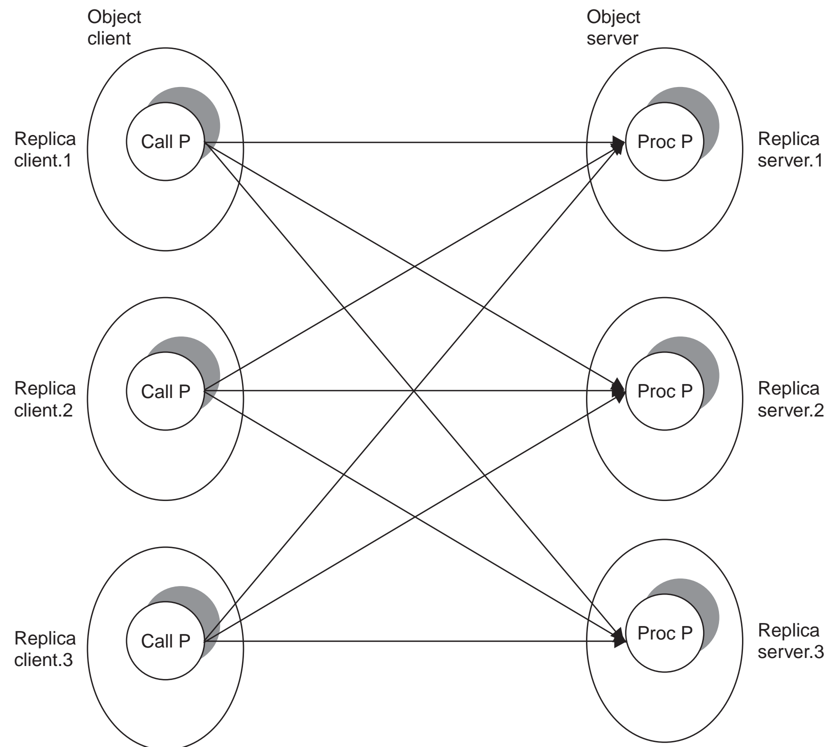

Figure 14.11 Replicated remote procedure calls.

There is clearly a trade-off between efficient use of the processing resources allocated to an object for availability and the time for it to recover from failures. Active replication is expensive in processing resources but requires little, if any, recovery time; in contrast warm replication (often called **passive** replication) is cheaper but has a slower recovery time. This has led to a compromise called **leader follower** replication (Barrett et al., 1990), where all replicas are active but one is deemed to be the primary version and takes all the decisions that are potentially non-deterministic. These decisions are then communicated to the followers. This form of replication is cheaper to support than active replication and does not have such a lengthy recovery time as passive replication.

Tolerating processor failure through dynamic redundancy

One of the problems of providing fault tolerance transparently to the application is that it is impossible for the programmer to specify degraded or safe execution. The alternative to static redundancy and replication is to allow processor(s) failure to be

handled dynamically by the application programmer. Clearly, the techniques that have been discussed so far in this book which enable an application to tolerate software design faults (for example, atomic actions) will provide some measure of tolerance of hardware faults. In Chapter 5, the four phases of fault tolerance were described: error detection, damage confinement and assessment, error recovery and fault treatment with continued service. In the context of processor failure, the following actions must be performed:

(1) The failure of the processor(s) must be detected and communicated to the remaining processors in the system. Failure detection would normally be done by the underlying distributed run-time support system. However, there must be an appropriate way of communicating which processors have failed to the application software. Fault-Tolerant Concurrent C provides such a notification mechanism.

(2) The damage that has occurred because of the failure must be assessed; this requires knowledge of which processes were running on the failed processor(s), which processors and processes remain active (and their state). For this to be achieved, the application programmer must have control over which objects are placed on which machines. Furthermore, it must be clear what effect a processor failure has on the processes executing on the failed processor, and their data. Also, the effect of any interaction, either pending or future, with those processes and their data must be defined.

(3) Using the results of the damage assessment, the remaining software must agree on a response to the failure and carry out the necessary actions to effect that response. To achieve maximum tolerance, this part of the recovery procedure must be distributed. If only a single processor performs the recovery operations then failure of that processor will be catastrophic to the application. Recovery will require communication paths between processes being changed so that alternative services can be provided. Also, because the response selected will depend on the overall state of the application, it will be necessary for certain data to be made available on all machines and for this to be held in a consistent state. For example, the actions to be taken following a processor failure in an avionics system will depend on the altitude of the plane. If different processors have different values for this altitude then the chosen recovery procedures may work at cross purposes.

(4) As soon as is practicable, the failed processor and/or its associated software must be repaired and the system returned to its normal error-free state.

None of the real-time programming languages considered in this book provides adequate facilities to cope with dynamic reconfiguration after processor failure.

Ada and fault-tolerance

The Ada language makes no assumption about the failure model underlying the implementation of programs. All that is specified is that a pre-defined exception `Communication_Error` is raised if one partition attempts to communicate with another and an error is detected by the communication subsystem.

Ada also does not support any particular approach to fault tolerance but allows an implementation to provide appropriate mechanisms. The following discussion assumes that a distributed implementation of Ada is executed on top of fail-stop processors.

The ability to replicate partitions is also not explicitly provided by Ada (although an implementation is free to do so – for example see (Wolf, 1998)). However, the Partition Communication Subsystem (PCS) can be extended, so a replicated remote procedure call facility could be provided. Each replicated partition has an associated group identifier which can be used by the system. All remote procedure calls are potentially replicated calls. The body of the package would require access to an ordered multicast facility. Note, however, that this approach does not allow arbitrary partitions to be replicated with the full Ada run-time system being involved at every scheduling decision. Hence, further restriction would be required.

Asynchronous notification could be provided by providing a protected object in an extended PCS. The run-time system could then call a procedure when it detects processor failure. This can open an entry which can then cause an asynchronous transfer of control in one or more tasks. Alternatively, a single (or group of) task(s) can be waiting for a failure to occur. The following package illustrates the approach.

```
package System.RPC.Reliable is

  type Group_Id is range 0 .. implementation_defined;

  -- Synchronous replicated call
  procedure Do_Replicated_RPC(Group : in Group_Id;
            Params : access Params_Stream_Type;
            Results : out Param_Stream_Access);

  -- Asynchronous call
  procedure Do_Replicated_APC(Group : in Group_Id;
            Params : access Params_Stream_Type);

  type RRPC_Receiver is access procedure (Service : in Service_Id;
            Params : access Params_Stream_Type;
            Results : out Param_Stream_Access);

  procedure Establish_RRPC_Receiver(Partition : in Partition_Id;
            Receiver : in RRPC_Receiver);

  protected Failure_Notify is
    entry Failed_Partition(P : out Partition_Id);
  private
    procedure Signal_Failure(P : Partition_Id);
    ...
  end Failure_Notify;

private
      ...
end System.RPC.Reliable;
```

Once tasks have been notified, they must assess the damage that has been done to the system. This requires knowledge of the actual configuration. Reconfiguration is

possible by using the dynamic binding facilities of the language (see Burns and Wellings (1998)).

Achieving reliable execution of occam2 programs

Although occam2 was designed for use in a distributed environment, it does not have any failure semantics. Processes which fail due to internal error (for example, array bound error) are equivalent to the STOP process. A process which is waiting for communication on a channel where the other process is on a failed processor will wait forever, unless it has employed a timeout. However, it is possible to imagine that suitable occam2 semantics for this event would be to consider both processes as STOPped and to provide a mechanism whereby some other process can be informed.

Achieving reliable execution of Real-Time Java programs

Java requires that each method that can be called remotely declares `RemoteException` in its throws list. This allows the underlying implementation to give notification when a remote call fails. Furthermore, Java leaves open the possibility of defining remote services which support replication. Currently, Real-Time Java is silent on its use in a distributed system.

14.6 Distributed algorithms

So far, this chapter has concentrated on the expression of distributed programs in languages like Ada, Java and occam2, along with the general problems of tolerating processor and communication failure. It is beyond the scope of this book to consider the specific distributed algorithms which are required for controlling and coordinating access to resources in a distributed environment. However, it is useful to establish certain properties that can be relied on in a distributed environment. In particular, it is necessary to show how events can be ordered, how storage can be organized so that its contents survive a processor failure, and how agreement can be reached in the presence of faulty processors. These algorithms are often needed to implement atomic multicast protocols.

14.6.1 Ordering events in a distributed environment

In many applications, it is necessary to determine the order of events that have occurred in the system. This presents no difficulty for uniprocessor or tightly coupled systems, which have a common memory and a common clock. For distributed systems, however, there is no common clock and the delay which occurs in sending messages between processors means that these systems have two important properties:

- For any given sequence of events, it is impossible to prove that two different processes will observe, identically, the same sequence.
- As state changes can be viewed as events, it is impossible to prove that any two processes will have the same global view of a given subset of the system state. In Chapter 12, the notion of causal ordering of events was introduced to help solve this problem.

If processes in a distributed application are to coordinate and synchronize their activities in response to events as they occur, it is necessary to place a causal order on these events. For example, in order to detect deadlock between processes sharing resources, it is important to know that a process released resource A before requesting resource B. The algorithm presented here enables events in a distributed system to be ordered and is due to Lamport (1978).

Consider process P which executes the events $p0$, $p1$, $p2$, $p3$, $p4$ ... pn. As this is a sequential process the event $p0$ must have happened before event $p1$ which must have happened before event $p2$ and so on. This is written as $p0 \rightarrow p1 \rightarrow p2 \rightarrow ... \rightarrow pn$. Similarly for process $Q : q0 \rightarrow q1 \rightarrow q2 \rightarrow ... \rightarrow qn$. If these two processes are distributed and there is no communication between them, it is impossible to say whether $p0$ happened before or after $q0$, $q1$ and so on. Consider now the case where the event $p1$ is sending a message to Q, and $q3$ is the event which receives the message from P. Figure 14.12 illustrates this interaction.

As the act of receiving a message must occur *after* the message has been sent, then $p1 \rightarrow q3$, and as $q3 \rightarrow q4$, it follows that $p1 \rightarrow q4$ (that is, $p1$ could have a causal effect on $q4$). There is still no information as to whether $p0$ or $q0$ happened first. These events are termed **concurrent events** (as there is no causal ordering). As neither event can affect the other, it is of no real importance which one is considered to have occurred first. However, it is important that processes which make decisions based on this order all assume the same order.

To order all events totally in a distributed system, it is necessary to associate a *time-stamp* with each event. However, this is not a physical time-stamp but rather a logical time-stamp. Each processor in the system keeps a logical clock, which is incremented every time an event occurs on that processor. An event $p1$ in process P occurred before $p2$ in the same process if the value of the logical clock at $p1$ is less than the value of the logical clock at $p2$. Clearly, it is possible using this method for the logical clock associated with each process to get out of synchronization. For example, P's logical clock at $p1$ may be greater than Q's logical clock at $q3$; however, $p1$ must have occurred before $q3$ because a message cannot be received before it has been sent. To resolve this problem, it is necessary for every message sent between processes to carry the time-stamp of the event that sent the message. Furthermore, every time a process receives a message it must set its logical clock to a value which is the greater of its own value and:

- the time-stamp found in the message plus at least one – for asynchronous message passing; or
- the time-stamp – for synchronous message passing.

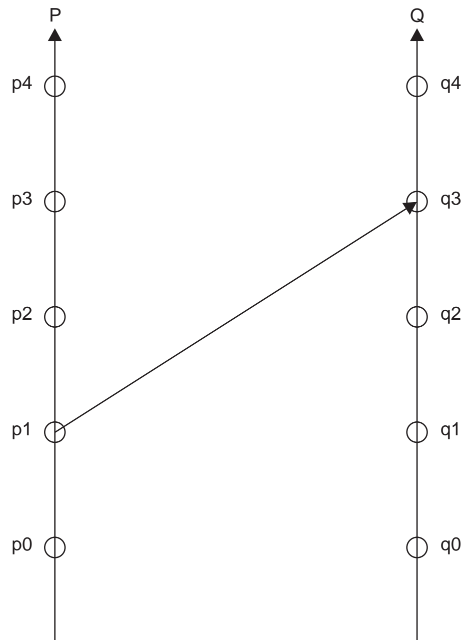

Figure 14.12 Two interacting processes.

This ensures that no message is received before it is sent.

Using the above algorithm, all events in the system have an associated time-stamp and can be partially ordered accordingly. If two events have the same time-stamp then it is sufficient to use an arbitrary condition, such as the numeric value of process identifiers, to give an artificial total order to the events.

14.6.2 Implementing global time

The approach to global ordering described above used logical clocks. An alternative scheme is to use a global time model based on physical time. This could be achieved by all nodes having access to a single time source, but it is more usual for nodes to have their own clocks. Hence it is necessary to coordinate these local clocks. There are many algorithms for doing this. All involve compensating for the inevitable clock drift that occurs even when clocks are notionally identical.

With quartz crystals, two clocks may drift apart by over one second in approximately six days. If timing events are significant at the millisecond level, drift becomes a problem after only eight minutes.

To bound clock drift, it is necessary to manipulate the time base at each node. Time must never move backwards and hence, although slow clocks can jump forward a

defined number of ticks, fast clocks must be slowed down rather than be moved backwards in time.

A clock coordination algorithm will provide a bound, Δ, on the maximum difference between any two clocks (as would be perceived by an external reference clock). Event A can now be assumed to precede event B if (and only if):

$$t(A) \ + \ \Delta \ < \ t(B)$$

where $t(A)$ is the time of event A.

One way to achieve synchronization between nodes is for a central time server process, S, to supply the time according to its own clock upon request. The time server may be linked to an external time source or have a more accurate hardware clock. Alternatively, it may be an arbitrary node in the system that is deemed to be the source of system time. In a distributed system, communications are not instantaneous, and hence the time provided by S is subject to two sources of error – variability in the time taken for the message to be returned from S, and some non-determinism introduced by the responsiveness of the client once it receives the message. It is also important that S is not preempted between readings its clock and sending out the message.

To reduce the jitter introduced by the message round trip, S can periodically send out its 'time'. If, additionally, a high-priority message is used (and S itself has a high priority) then the other sources of variability can be minimized.

An alternative approach is for the client of S to send its own time reading with its message. Upon the receipt of a message, S looks at its clock and calculates a correction factor, δ:

$$\delta = t(client) + min - t(S)$$

where min is the minimum communication delay for the message from the client.

The correction factor is then transmitted back to the client. The transmission time of this second message is not time-critical. Upon receipt of this second message, the client will either advance its own clock by δ if its value is negative, or slow down its clock by this amount if δ is positive. The use of a single time source gives a simple scheme but is subject to the single point failure of S (or the node on which it is executing). Other algorithms use a decentralized approach whereby all nodes broadcast their 'time' and a consensus vote is taken. Such a vote may omit outlying values. These clock synchronization algorithms must satisfy both an **agreement** and an **accuracy** property, where:

- the agreement condition is satisfied only if the skew between any two non-faulty clocks is bounded; and
- the accuracy condition is satisfied only if all non faulty clocks have a bounded drift with respect to real time.

14.6.3 Implementing stable storage

In many instances, it is necessary to have access to storage whose contents will survive a processor crash; this is called **stable storage**. As the main memory of any processor is volatile, it is necessary to use a disk (or any other form of non-volatile storage) as the stable storage device. Unfortunately, write operations to disks are not atomic in that the operation can crash part way through. When this happens, it is not possible for a recovery manager to determine whether the operation succeeded or not. To solve this problem, each block of data is stored twice on separate areas of the disk. These areas are chosen so that a head crash while reading one area will not destroy the other; if necessary, they can be on physically separate disk drives. It is assumed that the disk unit will indicate whether a single write operation completes successfully (using redundancy checks). The approach, in the absence of a processor failure, is to write the block of data to the first area of the disk (this operation may have to be repeated until it succeeds); only when this succeeds is the block written to the second area of the disk.

If a crash occurs while updating stable storage the following recovery routine can be executed.

```
read_block1;
read_block2;
if both_are_readable and block_1 = block2 then
    -- do nothing, the crash did not affect the stable storage
else
    if one_block_is_unreadable then
        -- copy good block to bad block
    else
        if both_are_readable_but_different then
            -- copy block1 to block2 (or visa versa)
        else
            -- a catastrophic failure has occurred and both
            -- blocks are unreadable
        end
    end;
end;
```

This algorithm will succeed even if there are subsequent crashes during its execution.

14.6.4 Reaching agreement in the presence of faulty processes

Early on in this chapter it was assumed that if a processor fails it fails silently. By this, it is meant that the processor effectively stops *all* execution and does not take part in communication with any other processors in the system. Indeed, even the algorithm for stable storage presented above assumes that a processor crash results in the processor immediately stopping its execution. Without this assumption, a malfunctioning processor might perform arbitrary state transitions and send spurious messages to other processors. Thus even a logically correct program could not be guaranteed to produce

the desired result. This would make fault-tolerant systems seemingly impossible to achieve. Although every effort can be made to build processors that operate correctly in spite of component failure, *it is impossible* to guarantee this using a *finite* amount of hardware. Consequently, it is necessary to assume a bounded number of failures. This section considers the problem of how a group of processes executing on different processors can reach a consensus in the presence of a bounded number of faulty processes within the group. It is assumed that all communication is reliable (that is, sufficiently replicated to guarantee reliable service).

Byzantine generals problem

The problem of agreeing values between processes which may reside on faulty processors is often expressed as the **Byzantine Generals Problem** (Lamport et al., 1982). Several divisions of the Byzantine Army, each commanded by its own general, surround an enemy camp. The generals are able to communicate by messengers and must come to an agreement as to whether to attack the camp. To do this, each observes the enemy camp and communicates his or her observations to the others. Unfortunately, one or more of the generals may be traitors and liable to convey false information. The problem is for all loyal generals to obtain the same information. In general, $3m + 1$ generals can cope with m traitors if they have $m + 1$ rounds of message exchanges. For simplicity, the approach is illustrated with an algorithm for four generals that will cope with one traitor, and the following assumptions will be made:

(1) Every message that is sent is delivered correctly.

(2) The receiver of a message knows who sent it.

(3) The absence of a message can be detected.

The reader is referred to the literature for more general solutions (Pease et al., 1980; Lamport et al., 1982).

Consider general Gi and the information that he/she has observed Oi. Each general maintains a vector V of information he/she has received from the other generals. Initially Gi's vector contains only the value Oi; that is $Vi(j) = null(for\ i <> j)$ $and\ Vi(i) = Oi$. Each general sends a messenger to every other general indicating his or her observation; loyal generals will always send the correct observation; the traitor general may send a false observation and may send different false observations to different generals. On receiving the observations, each general updates his or her vector and then sends the value of the other three's observations to the other generals. Clearly, the traitor may send observations different from the ones he or she has received, or nothing at all. In the latter case, the generals can choose an arbitrary value.

After this exchange of messages, each loyal general can construct a vector from the majority value of the three values it has received for each general's observation. If no majority exists then they assume, in effect, that no observations have been made.

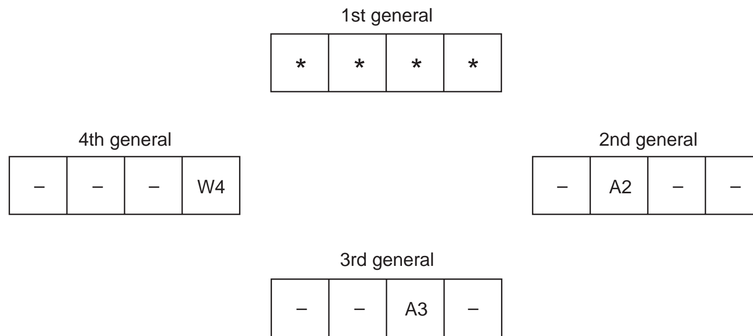

Figure 14.13 Byzantine generals – initial state.

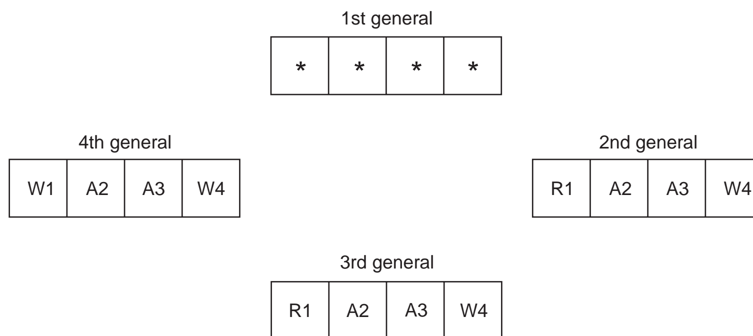

Figure 14.14 Byzantine generals – state after first message exchange.

For example, suppose that the observations of a general lead him or her to one of three conclusions: attack (A), retreat (R), or wait (W). Consider the case where $G1$ is a traitor, $G2$ concludes attack, $G3$: attack and $G4$: wait. Initially the state of each vector is given in Figure 14.13.

The index into the vector (1..4) gives the General number, the contents of the item at that index gives the observation from that general and who reported the observation. Initially then the fourth general stores 'Wait' in the fourth element of its vector indicating that the observation came from him- or herself. Of course, in the general case it might not be possible to authenticate who reported the observation. However, in this example it will be assumed that the traitor is not that devious.

After the first exchange of messages, the vectors are (assuming that the traitor realises the camp is vulnerable and sends the retreat and wait messages randomly to all

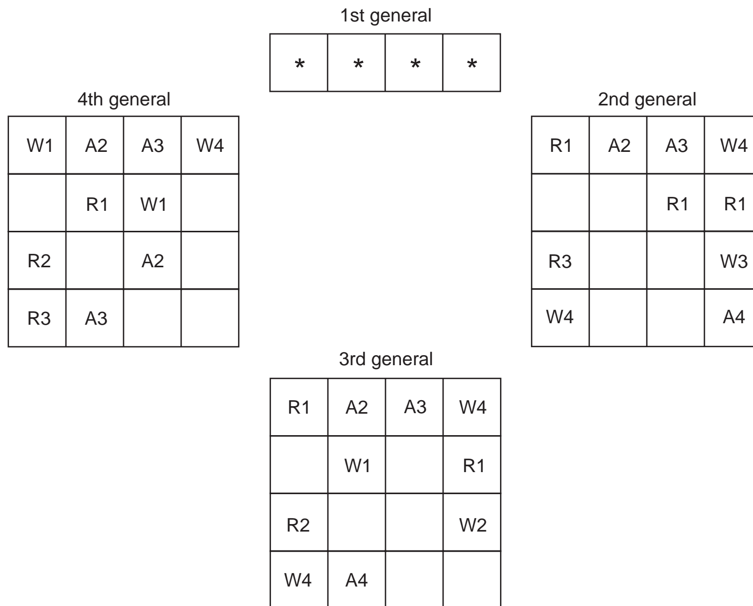

Figure 14.15 Byzantine generals – second message exchange

generals) shown in Figure 14.14. After the second exchange of messages, Figure 14.15 gives the information which is available to each general (again assuming the traitor sends retreat and wait messages randomly).

To illustrate how this table has been derived, consider the final row for the fourth general. This has been obtained, allegedly, from the third general (as the 3 indicates) and includes information about the decisions of the first and the second generals. Hence R3 in the first column indicates that the third general is of the view that the first general wishes to retreat.

Figure 14.16 gives a final (majority) vector. Loyal generals have a consistent view of each others' observations and, therefore, can make a uniform decision.

If it is possible to restrict the actions of the traitors further (that is, a stricter failure model), then the number of generals (processors) required to tolerate m traitors (failures) can be reduced. For example, if a traitor is unable to modify a loyal general's observation (say, he or she must pass a signed copy of the observation and the signature cannot be forged or modified), then only $2m + 1$ generals (processors) are required.

Using solutions to the Byzantine General problem, it is possible to construct a fail-silent processor by having internally replicated processors carry out a Byzantine agreement (Schneider, 1984). If the non-faulty processors detect a disagreement they stop execution.

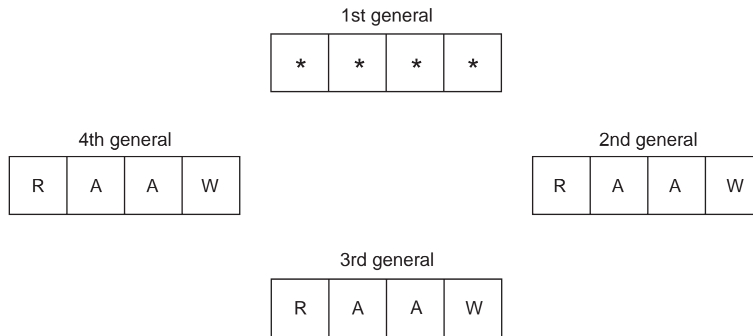

Figure 14.16 Byzantine generals – final state.

14.7 Deadline scheduling in a distributed environment

Having considered a number of distributed architectures, communication protocols and algorithms, it is now possible to return to the key issue of understanding the temporal behaviour of applications built upon a distributed environment. Here, parts of the system can be making progress at different rates and communication delays become significant. Not only must the time of the environment be linked to that of the computer system, but the different processors/nodes need to have some form of time linkage. The term **synchronous** is used (in this context) to designate a distributed system that has the following properties:

- There is an upper bound on message delays; this consists of the time it takes to send, transport and receive a message over some communications link.
- Every processor has a local clock, and there is a bounded drift rate between any two clocks.
- The processors themselves make progress at a minimum rate at least.

Note that this does not imply that faults cannot occur. Rather, the upper bound on message delay is taken to apply only when non-faulty processor are communicating over a non-faulty link. Indeed, the existence of the upper bound can be used to provide failure detection.

A system that does not have any of the above three properties is called **asynchronous**.

In this section only synchronous systems are considered. Two main topics will be investigated. First will be the issue of allocating processes to processors, and then the scheduling of communications will be discussed.

14.7.1 Allocation

The development of appropriate scheduling schemes for distributed (and multiprocessor) systems is problematic. Graham (1969) showed that multiprocessor systems can behave quite unpredictably in terms of the timing behaviour they exhibit. He used dynamic allocation (that is, processes are assigned to processors as they become runnable), and was able to illustrate the following anomalies:

- Decreasing the execution time of some process, P, could lead to it having an increased response time.
- Increasing the priority of process, P, could lead to it having an increased response time.
- Increasing the number of processors could lead to P having an increased response time.

All of these results are clearly counter-intuitive.

Mok and Dertouzos (1978) showed that the algorithms that are optimal for single processor systems are not optimal for increased numbers of processors. Consider, for example, three periodic processes P_1, P_2 and P_3 that must be executed on two processors. Let P_1 and P_2 have identical deadline requirements, namely a period and deadline of 50 time units and an execution requirement (per cycle) of 25 units; let P_3 have requirements of 100 and 80. If the rate monotonic algorithm (discussed in Chapter 13) is used, P_1 and P_2 will have highest priority and will run on the two processors (in parallel) for their required 25 units. This will leave P_3 with 80 units of execution to accomplish in the 75 units that are available. The fact that P_3 has two processors available is irrelevant (one will remain idle). As a result of applying the rate monotonic algorithm, P_3 will miss its deadline even though average processor utilization is only 65%. However, an allocation that maps P_1 and P_2 to one processor and P_3 to the other easily meets all deadlines.

Other examples can show that the earliest deadline first (EDF) formulation is similarly non-optimal. This difficulty with the optimal uniprocessor algorithms is not surprising as it is known that optimal scheduling for multiprocessor systems is NP-hard (Graham et al., 1979). It is, therefore, necessary to look for ways of simplifying the problem and to provide algorithms that give adequate sub-optimal results.

Allocation of periodic processes

The above discussion showed that judicious allocation of processes can significantly affect schedulability. Consider another example; this time let four processes be executing on the two processors, and let their cycle times be 10, 10, 14 and 14. If the two 10s are allocated to the same processor (and by implication the two 14s to the other) then 100% processor utilization can be achieved. The system is schedulable even if execution times for the four processes are 5, 5, 10 and 4 (say). However, if a 10 and a 14 were placed together on the same processor (as a result of dynamic allocation) then maximum utilization drops to 83%.

What this example appears to show is that it is better to allocate periodic processes statically rather than let them migrate and, as a consequence, unbalance the allocation and potentially downgrade the system's performance. Even on a tightly coupled system running a single run-time dispatcher, it is better to keep processes on the same processor rather than try to utilize an idle processor (and risk unbalancing the allocation).

If static deployment is used, then the deadline monotonic algorithm (or other optimal uniprocessor schemes) can test for schedulability on each processor. In performing the allocation, processes that are harmonically related should be deployed together (that is, to the same processor) as this will help increase utilization.

Allocation of sporadic and aperiodic processes

As it appears expedient to allocate periodic processes statically, a similar approach to sporadic processes would seem to be a useful model to employ. If all processes are statically mapped then the algorithms discussed in Chapter 13 can be used on each processor (that is, each processor, in effect, runs its own scheduler/dispatcher).

To calculate execution times (worst case or average) requires knowledge of potential blocking. Blocking within the local processor can be bounded by inheritance or ceiling protocols. However, in a multiprocessor system there is another form of blocking: this is when a processes is delayed by a process on another processor. This is called remote blocking and is not easily bounded. In a distributed system, remote blocking can be eliminated by adding processes to manage the distribution of data. For example, rather than be blocked waiting to read some data from a remote site, an extra process could be added to the remote site whose role is to forward the data to where it is needed. The data is thus available locally. This type of modification to a design can be done systematically; however, it does complicate the application (but does lead to a simpler scheduling model).

One of the drawbacks of a purely static allocation policy is that no benefits can be gained from spare capacity in one processor when another is experiencing a transient overload. For hard real-time systems, each processor would need to be able to deal with worst-case execution times for its periodic processes, and maximum arrival times and execution times for its sporadic load. To improve on this situation Stankovic et al. (1985) and Ramamritham and Stankovic (1984) have proposed more flexible task scheduling algorithms.

In their approach, all safety critical periodic and sporadic processes are statically allocated but non-critical aperiodic processes can migrate. The following protocol is used:

- Each aperiodic process arrives at some node in the network.
- The node at which the aperiodic process arrives checks to see if this new process can be scheduled together with the existing load. If it can, the process is said to be guaranteed by this node.
- If the node cannot guarantee the new processes, it looks for alternative nodes that may be able to guarantee it. It does this using knowledge of the state of the whole network and by bidding for spare capacity in other nodes.

- The process is thus moved to a new node where there is a high probability that it will be scheduled. However, because of race conditions, the new node may not be able to schedule it once it has arrived. Hence the guarantee test is undertaken locally; if the process fails the test then it must move again.
- In this way, an aperiodic process is either scheduled (guaranteed) or it fails to meet its deadline.

The usefulness of their approach is enhanced by the use of a linear heuristic algorithm for determining where a non-guaranteed process should move. This heuristic is not computationally expensive (unlike the optimal algorithm, which is NP-hard), but does give a high degree of success; that is, there is a high probability that the use of the heuristic will lead to an aperiodic process being scheduled (if it is schedulable at all).

The cost of executing the heuristic algorithm and moving the aperiodic processes is taken into account by the guarantee routine. Nevertheless, the scheme is workable only if aperiodic processes can be moved, and that this movement is efficient. Some aperiodic processes may be tightly coupled to hardware unique to one node and will have at least some component that must execute locally.

14.7.2 Scheduling access to communications links

Communication between processes on different machines in a distributed system requires messages to be transmitted and received on the underlying communication subsystem. In general, these messages will have to compete with each other to gain access to the network medium (for example, switch, bus or ring). In order for hard real-time processes to meet their deadlines, it will be necessary to schedule access to the communication subsystem in a manner which is consistent with the scheduling of processes on each processor. If this is not the case then priority inversion may occur when a high-priority process tries to access the communications link. Standard protocols such as those associated with Ethernet do not support hard deadline traffic, as they tend to queue messages in a FIFO order or use non-predictable back-off algorithms when there is a message collision.

Although the communication link is just another resource, there are at least four issues which distinguish the link scheduling problem from processor scheduling.

- Unlike a processor, which has a single point of access, a communications channel has many points of access – one for each attached physical node. A distributed protocol is therefore required.
- While preemptive algorithms are appropriate for scheduling processes on a single processor, preemption during message transmission will mean that the entire message will need retransmitting.
- In addition to the deadlines imposed by the application processes, deadlines may also be imposed by buffer availability – the contents of a buffer must be transmitted before new data can be placed in it.

Although many *ad hoc* approaches are used in distributed environments there are at least three schemes that do allow for predictable behaviour. These will now be briefly discussed.

TDMA

The natural extension to using a cyclic executive for uniprocessor scheduling, is to use a cyclic approach to communications. If all application processes are periodic, it is possible to produce a communications protocol that is slotted by time. Such protocols are call TDMA (Time Division Multiple Access). Each node has a clock that is synchronized to all other node clocks. During a communications cycle, each node is allocated time slots in which it can communicate. These are synchronized to the execution slots of each node's cyclic executive. No message collisions can occur, as each node knows when it can write and, moreover, each node knows when there is a message available that it needs to read.

The difficulty with the TDMA approach come from constructing the schedule. This difficulty increases exponentially with the number of nodes in the system. One architecture that has shown considerable success in using TDMA is the TTA (Time Triggered Architecture) (Kopetz, 1997). It uses a complex graph reduction heuristic to construct the schedules. The other drawback of TDMA is that it is difficult to plan when sporadic messages can be communicated.

Timed token-passing schemes

One approach for generalizing away from a purely time-slotted approach is to use a token-passing scheme. Here a special message (the **token**) is passed from node to node. Nodes can send out messages only when they hold the token. As there is only one token, no message collisions can occur. Each node can hold the token only for a maximum time, and hence there is a bounded **token rotation time**.

A number of protocols use this approach. An example being the fibre optic FDDI (Fiber Distributed Data Interface) protocol. Here messages are grouped into two classes called, confusingly, synchronous and asynchronous. Synchronous messages have hard time constraints and are used to define each node's token-holding time and hence the *target token rotation time*. Asynchronous messages are not deemed to have hard constraints; they can be communicated by a node if either it has no synchronous messages to send or the token has arrived early (because other nodes have had nothing to transmit). The worst-case behaviour of this protocol occurs when a message arrives at a node just as the token is being passed on. Moreover, up to that time no node has transmitted messages, and hence the token is being delivered very early. After the token has been passed on from the node with the new synchronous message, the rest of the nodes now have lots of messages to send. The first node sends its synchronous load and, as the token arrived very early, sends a full set of asynchronous messages; all subsequent nodes send their synchronous load. By the time the token arrives back at the node of

interest, a time interval equal to twice the *target token rotation time* has passed. This is the bounded delivery time.

Priority-based protocols

Given the benefits that have been derived from using priority-based scheduling on processors, it is reasonable to assume that a priority-based approach to message scheduling would be useful. Such protocols tend to have two phases. In the first phase, each node indicates the priority of the message it wishes to transmit. This is obviously the maximum priority of the set of messages it may have outstanding. At the end of this first phase, one node has gained the right to transmit its message. The second phase is simply the communication of this message. In some protocols, the two phases can overlap (that is, while one message is being broadcast, parts of the message are modified so that the priority of the next message is determined).

While priority-based protocols have been defined for some time, they have tended to have the disadvantage that only a small range of priorities have been supported by communication protocols. They have been used to distinguish between broad classes of messages rather than for message scheduling. As indicated in Chapter 13, the best results of priority-based scheduling occur when each process (or message in this discussion) has a distinct priority. Fortunately, there are some protocols that do now provide for a large priority field. One example is CAN (Controller Area Network) (Tindell et al., 1995).

In CAN 2.0A, an 8 byte message is preceded by an 11 bit identifier that acts as a priority. At the start of a transmission sequence, each node writes (simultaneously) to the broadcast bus the first bit of its maximum priority message identifier. The CAN protocol acts like a large AND gate; if any node writes a 0 then all nodes read a 0. The 0 bit is said to be *dominant*. The protocol proceeds as follows (for each bit of the identifier):

- If a node transmits a zero it continues on to the next bit.
- If a node transmits a one and reads back a one then it continues on to the next bit.
- If a node transmits a one and reads back a zero, it backs off and takes no further part in this transmission round.

The lower the value of the identifier the higher the priority. As identifiers are unique, the protocol is forced to end up with just one node left in after the 11 rounds of bitwise arbitration. This node then transmits its message.

The value of CAN is that it is a genuine priority-based protocol and hence all the analysis presented in Chapter 13 can be applied. The disadvantage of the type of protocol used by CAN is that it restricts the speed of communication. In order for all nodes to write their identifiers' bits 'at the same time' and for them all to read the subsequent ANDed value (and act on this value before sending out, or not, the next bit) there must be a severe bound on the transmission speed. This bound is actually a function of the length of the wire used to transmit the bits, and hence CAN is not

suitable for geographically dispersed environments. It was actually designed for the informatics within modern automobiles – where it is having considerable success.

ATM

ATM can be used across both wide area and local area networks. The goal is to support a wide range of communication requirements such as those needed for voice and video as well as data transmission. ATM supports point-to-point communication via one or more switches. Typically, each computer in the network is connected to a switch via two optical fibres: one taking traffic to the switch, the other relaying traffic from the switch.

All data to be transmitted is divided into fixed-sized packets call **cells**, where each cell has a 5-byte header and a 48-byte data field. Applications communicate via **virtual channels** (VC). The data to be transmitted on a particular VC can have certain timing behaviour associated with it, such as its bit rate, period or its deadline. An **adaptation layer** provides specific services to support the particular class of user data; the precise behaviour of this layer is variable to suit the data transmission needs of a particular system. It is within the adaption layer that end-to-end error correction and synchronization, and the segmentation and reassembly of user data into the ATM cells are performed.

A typical ATM network will contain multiple switches, each of which has an associated cell queuing policy - the basic FIFO policy was employed in most early commercial switches, but some now cater for priority-based queuing. Within an individual switch, a set of connections may be established between a number of switch input ports and output ports according to a connection table. Queuing of cells may occur at input and/or output ports.

A common approach taken in a real-time ATM solution is as follows:

- Pre-define all VC timing requirements;
- Pre-define the route and hence the network resources allocated to each VC;
- Control the total bandwidth required by each VC at each network resource through which it is routed;
- Calculate the resulting delays and hence assess the feasibility of a particular allocation of VCs to the network hardware.

The control of bandwidth usage by each VC is required in order to avoid congestion in the network. The motivation for this is to support predictable behaviour of each VC and consequently the network as a whole. When congestion is present, ATM cells are at risk of being discarded (under the normal rules of the ATM layer protocol) and worst-case delays are difficult to predict without excessive pessimism (allowing for detection of lost cells and possible retransmission).

14.7.3 Holistic scheduling

A reasonably large distributed real-time system may contain tens of processors and two or three distinct communication channels. Both the processor and the communication subsystems can be scheduled so that the worst-case timing behaviour is predictable. This is facilitated by a static approach to allocation. Having analyzed each component of the system it is then possible to link the predictions together to check compliance with system-wide timing requirements (Tindell and Clark, 1994; Palencia Gutierrez and Gonzalez Harbour, 1998; Palencia Gutierrez and Gonzalez Harbour, 1999). In addressing this *holistic scheduling* problem, two important factors need to be taken into consideration:

- Will variability in the behaviour of one component adversely affect the behaviour of another part of the system?
- Will the simple summation of each component's worst-case behaviour lead to pessimistic predictions?

The amount of variability will depend upon the approach to scheduling. If a purely time-triggered approach is used (that is, cyclic executives linked via TDMA channels), there is little room for deviation from the repeating behaviour. However, if priority-based scheduling is employed on the processors and channels then there could be considerable variation. For example, consider a sporadic process released by the execution of a periodic process on another node. On average the sporadic process will execute at the same rate as the periodic process (say, every 50 ms). But the periodic process (and the subsequent communication message to release the sporadic) will not take place at exactly the same time each period. It may be that the process executes relatively late on one invocation but earlier on the next. As a result the sporadic may be released for the second time only 30 ms after its previous release. To model the sporadic as a periodic process with a period of 50 ms would be incorrect and could lead to a false conclusion that all deadlines can be satisfied. Fortunately, this variability in release time can be accurately modelled using the release jitter analysis give in Section 13.12.2.

Whereas the response time analysis introduced for single processor scheduling is necessary and sufficient (that is, gives an accurate picture of the true behaviour of the processor), holistic scheduling can be pessimistic. This occurs when the worst-case behaviour on one subsystem implies that less than worst case will be experienced on some other component. Often only simulation studies will allow the level of pessimism to be determined (statistically). Further research is needed to accurately determine the effectiveness of holistic scheduling.

A final issue to note with holistic scheduling is that support can be given to the allocation problem. Static allocation seems to be the most appropriate to use. But deriving the best static allocation is still an NP-hard problem. Many heuristics have been considered for this problem. More recently, search techniques such as simulated annealing and genetic algorithms have been applied to the holistic scheduling problem. These have proved to be quite successful; and can be easily extended to systems that are replicated for fault tolerance (where it is necessary to come up with allocations that assign replicas to different components).

Summary

This chapter defined a distributed computer system to be a collection of autonomous processing elements, cooperating in a common purpose or to achieve a common goal. Some of the issues which arise when considering distributed applications raise fundamental questions that go beyond simple aspects of implementation. They are: language support, reliability, distributed control algorithms and deadline scheduling.

Language support

The production of a distributed software system to execute on a distributed hardware system involves: **partitioning** – the process of dividing the system into parts (units of distribution) suitable for placement onto the processing elements of the target system; **configuration** – associating the partitioned components with particular processing elements in the target system; **allocation** – the actual process of turning the configured system into a collection of executable modules and downloading these to the processing elements; **transparent execution** – executing the distributed software so that remote resources are accessed in a manner which is independent of their location (usually via remote procedure call); and **reconfiguration** – changing the location of a software component or resource.

Occam2, although designed for use in a distributed environment, is fairly low-level; the process is the unit of partitioning; configuration is explicit in the language but neither allocation nor reconfiguration are supported. Furthermore, remote access to resources is not transparent.

Ada allows collection of library units to be grouped into 'partitions' which communicate via remote procedure calls and remote objects. Neither configuration, nor allocation nor reconfiguration are supported directly by the language.

Java allows objects to be distributed which can communication via remote procedure calls or via sockets. None of configuration, allocation or reconfiguration is supported directly by the language.

CORBA facilitates distributed objects and interoperability between applications written in different languages for different platforms supported by middleware software from different vendors.

Reliability

Although the availability of multiple processors enables the application to become tolerant of processor failure, it also introduces the possibility of new types of faults occurring which would not be present in a centralized single-processor system. In particular, multiple processors introduce the concept of a partial system failure. Furthermore, the communication media may lose, corrupt, or change the order of messages.

Communication between processes across machine boundaries requires layers of protocols so that transient error conditions can be tolerated. Standards have been defined to support these layers of protocols. The OSI Reference Model is a layered model consisting of the Application, Presentation, Session Transport, Network, Data Link and Physical layers. It was developed primarily for wide area networks to enable open access; wide area networks were characterized by low-bandwidth communication with high error rates. TCP/IP is another protocol reference model aimed primarily at wide area networks. Most distributed embedded systems will, however, use local area network technology and will be closed to the outside world. Local area networks are characterized by high-bandwidth communication with low error rates. Consequently, although it is possible to implement language-level interprocess communication using the OSI approach, in practice the expense is often prohibitive. Thus, many designers tailor the communication protocols to the requirements of the language (the application) and the communication medium. These are called lightweight protocols.

When groups of processes are interacting, it is necessary for communication to be sent to the whole group. A **multicast** communication paradigm provides such a facility. It is possible to design a family of group communication protocols, each of which provides a multicast communication facility with specific guarantees: **unreliable multicast** – no guarantees of delivery to the group is provided; **reliable multicast** – the protocol makes a best-effort attempt to deliver the message to the group; **atomic multicast** – the protocol guarantees that if one process in the group receives the message then all members of the group receive the message; **ordered atomic multicast** – as well as guaranteeing the atomicity of the multicast, the protocol also guarantees that all members of the group will receive messages from different senders in the same order.

Processor failure can be tolerated through static or dynamic redundancy. Static redundancy involves replicating application components on different processors. The degree of replication can be varied according to the importance of a particular component. One of the problems of providing fault tolerance transparently to the application is that it is impossible for the programmer to specify degraded or safe execution. The alternative to static redundancy is to allow processor failure to be handled dynamically by the application programmer. This requires: the failure of the processor to be detected and communicated to the remaining processors in the system; the damage that has occurred must then be assessed; using these results the remaining software must agree on a response and carry out the necessary actions to effect that response; and as soon as is practicable the failed processor and/or its associated software must be repaired and the system returned to its normal error free state. Few of the real-time programming languages provide adequate facilities to cope with dynamic reconfiguration after processor failure.

Distributed control algorithms

The presence of true parallelism in an application together with physically distributed processors and the possibility that processors and communication links may fail, require many new algorithms for resource control. The following algorithms were considered: event ordering, stable storage implementation, and Byzantine agreement protocols. Many distributed algorithms assume that processors are "fail-stop"; this means either they work correctly or they halt immediately a fault occurs.

Deadline scheduling

Unfortunately, the general problem of dynamically allocating processes to processors (so that system-wide deadlines are met) is computationally expensive. It is therefore necessary to implement a less flexible allocation scheme. One rigid approach is to deploy all processes statically or to allow only the non-critical ones to migrate.

Once processors have been allocated, it is necessary to schedule the communications medium. TDMA, timed token passing and priority-based scheduling are appropriate real-time communication protocols. Finally, end-to-end timing requirements must be verified by considering the holistic scheduling of the entire system.

Further reading

Brown, C. (1994) *Distributed Programming with Unix*. Englewood Cliffs, NJ: Prentice Hall.

Comer, D. E. (1999) *Computer Networks and Internets*. New York: Prentice Hall.

Coulouris, G. F., Dollimore, J. and Kindberg, T. (2000) *Distributed Systems, Concepts and Design*, 3rd Edition. Harlow: Addison-Wesley.

Farley, J. (1998) *Java Distributed Computing*. Sebastopol, CA: O'Reilly.

Harold, E. (1997) *Java Network Programming*. Sebastopol, CA: O'Reilly.

Halsall, F. (1995) *Data Communications, Computer Networks and OSI* 2nd Edition. Reading, MA: Addison-Wesley.

Hoque, R. (1998) *CORBA 3*. Foster City, CA: IDG Books Worldwide.

Horstmann, C. S. and Cornell, G. (2000) *Core Java 2, Volume II – Advanced Features*. Sun Microsystems.

Kopetz, H. (1997) *Real-Time Systems: Design Principles for Distributed Embedded Applications*: New York: Kluwer International.

Lynch, N. (1996) *Distributed Algorithms*. San Mateo, CA: Morgan Kaufmann.

Mullender, S. (ed.) (1993) *Distributed Systems*, 2nd Edition. Reading, MA: Addison-Wesley.

Tanenbaum, A. S. (1994) *Distributed Operating Systems*. Englewood Cliffs, NJ: Prentice Hall.

Tanenbaum, A. S. (1998), *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall.

Exercises

14.1 Discuss some of the *disadvantages* of distributed systems.

14.2 Ada supports a Real-Time Systems Annex and a Distributed Systems Annex. Discuss the extent to which these two annexes constitute a Distributed Real-Time Systems Annex.

14.3 From a data abstraction viewpoint, discuss why variables should not be visible in the interface to a remote object.

14.4 Consider the implications of having timed and conditional entry calls in a distributed Ada environment.

14.5 The following occam2 process has five input channels and three output channels. All integers received down the input channels are output to all output channels:

```
INT I,J,temp:
WHILE TRUE
  ALT I = 1 FOR 5
    in[I] ? temp
      PAR J = 1 FOR 3
        out[J] ! temp
```

Because this process has an eight channel interface it cannot be implemented on a single transputer unless its client processes are on the same transputer. Transform the code so that it can be implemented on three transputers. (Assume a transputer has only four links.)

14.6 Compare and contrast the Ada, Java and CORBA remote object models.

14.7 To what extent can CORBA's Message Passing Services be implemented in Java?

14.8 Sketch the layers of communication that are involved when the French delegate at the United Nations Security Council wishes to talk to the Russian delegate. Assume that there are interpreters who translate into a common language (say English) and then pass on the message to telephone operators. Does this layered communication follow the ISO OSI model?

14.9 Why do the semantics of remote procedure calls differ from the semantics of ordinary procedure calls?

14.10 If the OSI network layer is used to implement an RPC facility, should a datagram or a virtual circuit service be used?

14.11 Compare and contrast the stable storage and replicated data approaches for achieving reliable system data which will survive a processor failure.

14.12 Redo the Byzantine Generals problem given in Section 14.6.4 with G1 as the traitor, G2 concluding retreat, G3 concluding wait and G4 concluding attack.

14.13 Given the choice between an Ethernet and a token ring as the real-time communication subsystem, which should be used if deterministic access under heavy loads is the primary concern?

14.14 Update the algorithm given in Section 10.8.2 for forward error recovery in a distributed system