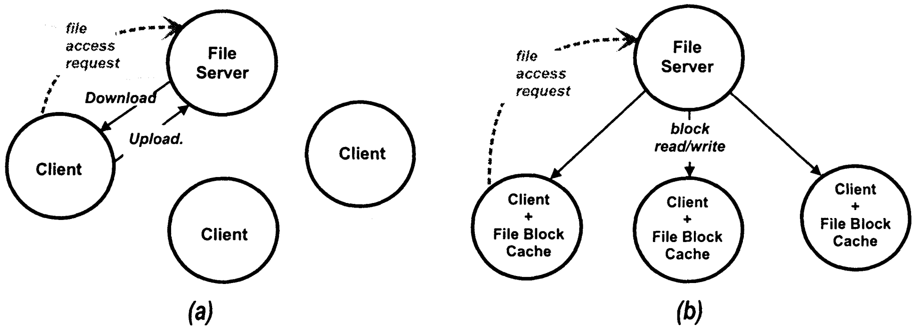


Figure 4.8. Distributed File Models: (a) Download-Upload; (b) Remote Access

are going to briefly review some successful systems, that have answered the previous questions in different ways.

4.2.1 NFS

The Network File System (NFS), developed by Sun Microsystems in 1984, was the first commercial distributed file system. It is a striking example of good distributed systems design.

The NFS is built using a client-server approach. Servers store files and respond to requests from remote applications that need to access those files. However, the applications are not requested to contact the server directly. Instead, file system calls from the application are redirected to a proxy of the remote service that executes on the client's machine. This proxy is simply called the *NFS client*, and it takes care of the interaction with the remote server. This interaction is performed using SUN's remote procedure call service. The design decision of using RPC to access the server, along with the strategic option of making the server's interface public, was one of the reasons that made NFS so popular. Since the interfaces and the format of the associated messages were known, it was possible for different companies to develop NFS components for many different architectures and operating systems.

The use of the client-server approach is made transparent to the client application through the addition of a Virtual File System (VFS) layer to the Unix kernel. The layer introduces an additional level of indirection in the file system calls. Instead of calling a specific file system primitive, the application calls the VFS interface that, in turn, calls the NFS client primitives. The VFS allows the kernel to support several file systems simultaneously, including the NFS and the native file system.

The NFS server uses a *stateless* approach, i.e., servers do not keep any state about open files. Also, servers keep no information about the number and state of their clients. This means that each request must be self-contained, *i.e.*, the server must be able to process the request just looking at its parameters, without any knowledge from past requests. Since the server remembers nothing, it does not lose any relevant information when it crashes. Thus, a stateless approach has some advantages from the fault tolerance point of view: a server may crash, recover, pick a new request and continue as if nothing happened (we postpone a further discussion about fault tolerance issues to the next part of the book). On the other hand, since the server does not keep state, it is unable to check if a file is being accessed by one or more clients. This, as we will see, makes the task of preserving the Unix semantics of file sharing impossible. Due to this reason, NFS does not support the *open* primitive (open semantics require the system to "remember" that the file has been opened). Instead, a *lookup* primitive, that provides a handle for a file name is provided.

A partial list of the NFS server interface is presented in Table 4.1. The reader will notice that *read* and *write* calls include an *offset* parameter. This is required because the server, being stateless, does not store the file pointer on behalf of the client. Instead, the file pointer must be stored by the NFS client and sent explicitly on read and write calls. The *cookie* parameter in the `readdir` call plays a similar role, allowing a client to read a large directory in pieces (the cookie points to the next directory entry to be read). The *lookup* primitive returns a *file handle* that consists of a pair: a unique *file system* identifier, and a unique *file* identifier. The unique file identifier is made of the Unix inode[1] of the file and a generation number, which is incremented whenever the inode is re-used. This ensures that identifiers are not reusable, and that the identifier of a new file cannot be confused with the identifier of a previous file on the same file system.

Table 4.1. NFS Interface (partial)

Name (parameters)	**Returns**
lookup (dirfh, name)	(fh, attr)
create (dirfh, name , attr)	(newfh, attr)
remove (dirfh, name)	(status)
read (fh, offset, count)	(attr, data)
write (fh, offset, count, data)	(attr)
mkdir (dirfh, name, attr)	(newfh, attr)
readdir (dirfh, cookie, count)	(direntries)

Consider now an application that reads a file, reading only one byte at a time. If a remote procedure call was to be performed for each byte read, the performance of the system would be unacceptable. This problem is not specific of distributed systems, in a centralized system one also avoids performing an I/O operation for each byte read by caching one or more file blocks in main

[1]In the Unix file system, files are uniquely identified by a index node, or simply, inode.

memory. In the NFS a similar approach can be followed, by caching the block both in the server's memory and in the client's memory. Unfortunately, having the same data cached in different machines introduces the problem of cache coherence. In a centralized system, the cache is physically shared by all processes. Since a single copy of the cached data exists, the sharing semantics of a centralized Unix file system is the following: if a process does a write, the results become immediately visible to all other processes. Clearly this is very expensive to obtain in a distributed system, since propagating an update requires at least one remote procedure call. The NFS approach implements a weaker consistency model that tries to balance consistency with performance requirements.

When reading a file, the NFS client reads a complete disk block (which is 8 kilobytes in the Unix BSD 4.2 Fast File System). The block is cached in the client's memory and is considered valid for some amount of time (typically, 3 seconds for files). Thus, subsequent reads that fall into the same block do not require a remote access to the server. After this period, a new access must first check with the server if the cache is still valid. For this purpose, the client also remembers the "version" it has cached, more precisely, the last time the data has been updated in the server. If the cache is still valid, it is assumed valid for another 3 seconds period; if not, the new version of the block is fetched again from the server.

Writes are executed in a similar manner. Instead of contacting the server each time a byte is written, the client caches all the writes for some time. The cache has to be flushed if the file is closed or if a *sync* call is performed by the application. Otherwise, updates are sent asynchronously to the server, using periods of low activity in the client. This task is performed by a Unix *daemon*, called the *block io daemon* (or simply the *bio-daemon*). The daemon can also try to optimize reads by performing *read-ahead*, i.e., requesting in anticipation the next block of a file being read by an application. To ensure that writes are guaranteed to be stored on disk when the remote procedure call returns, the cache on the server operates in *write-through* mode, *i.e.*, writes are immediately forced to disk.

So far, we have presented the interaction between a client and a server. We have not discussed how the client finds the appropriate server in the first place. The name of the servers storing remote files is configured at each client using an extension to the Unix *mount* facility. The mount mechanism allows a file system to be "attached" to another file system at a given directory, called the mount point. The NFS mount procedure allows a sub-tree of the server's file system to be mounted on a specific directory of the client machine (if the client has no disk, it can be mounted on the root directory of the client machine). When performing the mount operation the client contacts the server, which checks access rights and, in case of success, returns to the client a file handle to the mounted directory. If when translating the textual file name into a file handle, the NFS client traverses a mount point, it uses the file handle returned by the mount operation to perform the subsequent lookups.

4.2.2 AFS

The Andrew File System (AFS) was originally developed at Carnegie-Mellon University and later became a product of Transarc. The major design goal of AFS was to achieve scalability in terms of number of clients. Many of the design decisions behind the development of AFS aimed at overcoming known limitations of previous systems. For instance, with regard to the NFS file system described above, it was observed that the cache validation procedure, where clients inquire the servers about the validity of cached data, could easily overload the server with too many requests. Furthermore, it was noticed that most of these requests were unnecessary, given that the majority of files are not shared and thus, caches are usually valid.

To support their design decisions, the AFS development team made extensive measurements of file usage on their academic environment. The observations made at that time and in those environments have shown interesting facts: files were usually small; reads were much more common than writes and typically sequential; most files were updated by a single user; and when one file was accessed it was likely to be accessed again in the near future.

To address these access patterns, and assuming that local disks were available at client machines, a file system based on *whole file caching* was proposed, i.e., in AFS clients cache complete files (this approach has later been relaxed to accommodate very large files, by supporting caching of file portions). Once a file is cached, all read and write accesses are purely local and require no synchronization with the server. Once closed, the file remains in cache. When re-opened, the local cache is used whenever possible.

Enough disk space should be reserved in the clients' cache to hold the files needed for the typical operation of most applications. A daemon process in the client, called *Venus* is responsible for managing the AFS cache and for transferring files from and back to the server. The counterpart of Venus on the server is called *Vice*. To the clients, the file system appears seamless, though some files are local and others shared through Venus. This architecture is illustrated in Figure 4.9.

AFS supports read-write files accessed in competition by clients, but a RW file can have several read-only copies hosted at more than one Vice. Whenever the master file is updated, a *release* command makes sure that the RO copies are also updated. The above characteristics make AFS well-suited for a few classes of applications over file systems:

- shared read-only repositories, with occasional updates (e.g., news, price lists)– typically using RO copies for wider availability to many readers, single-client updates;
- shared read-write repositories, with infrequent updates (e.g., cooperative editing)– competitive few-writer activity, local caches remain valid for long;
- non-shared repositories (e.g., personal files)– single-writer activity, local cache remains valid wherever user is.

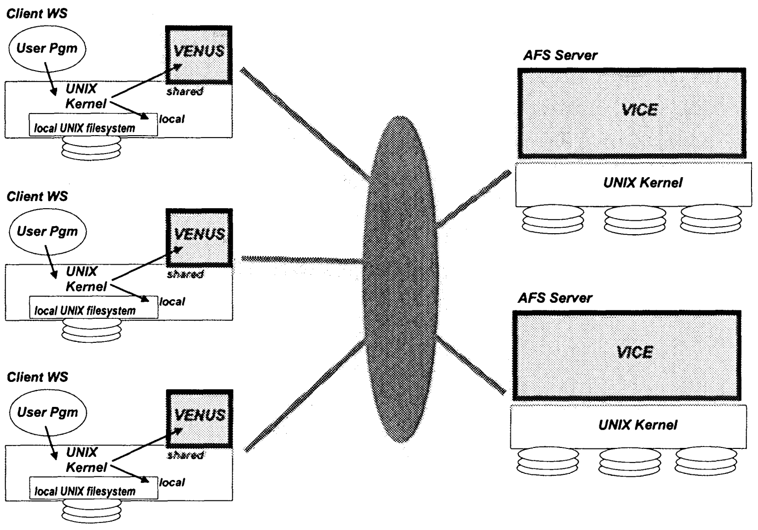


Figure 4.9. AFS Architecture

The goal of whole file caching is to relieve the server from unnecessary load. But, if the client should try to limit the interaction with a server to a minimum, how can it check the validity of data in its cache? The AFS approach consists in delegating on the server the responsibility for invalidating the client's cache when some other client updates the same file (actually, this optimization was only implemented in the second version of AFS).

When Vice gives Venus a copy of a file, it also makes a *callback promise*, or CBP. A CBP remains valid until hearing otherwise from Vice. When a file is changed at Vice, it calls back all clients holding valid CBPs for that file, so that they cancel the CBP, which invalidates the file cache. When Venus opens a file, it analyzes the cache. If the file is not in cache, it requests a copy to Vice. If the file is in cache, it checks the CBP. If the CBP is valid, it opens the local copy; if not, it requests a current copy to Vice.

The reader will note that there is a window during which a local copy may be opened that is not the most recent one[2]. In order to reduce this risk, an expiration mechanism exists that supersedes the algorithm described above: a file is only opened locally if it is less than T since the local Venus has last heard from Vice concerning this file. That is, if a file is opened after T, the latest version is downloaded from Vice. A typical value for T is 10 minutes. This mechanism also recovers from the loss of callback messages.

If the client crashes it may miss one or several callbacks from the server. Thus, when a client recovers it has to contact the server and determine the

[2]Erlier versions of AFS checked directly with Vice before opening, instead of the CBP, but this did not scale well.

status of all the files it holds, by checking the timestamps of the relevant CBPs with the file information on the server. To prevent the server from storing callback information indefinitely, access rights (also called the *file tokens*) are only valid for some limited period.

The semantics of AFS is not exactly one-copy. When more than one client open a file concurrently, the server will hold the state of the last file to be closed. This form of consistency is however adequate for the example classes we have enumerated earlier. Furthermore, applications can always superimpose their synchronization on top of these basic mechanisms.

4.2.3 Coda

The Coda file system is a follow-up of the AFS project at CMU lead by some members of the AFS development team. The main goals of Coda were to improve reliability and availability vis-a-vis partitioning, and to support nomadic and mobile computing. This was achieved by *whole volume replication*, and by *disconnected operation*. Coda supports the use of portable computers as file system clients, and tries to offer what the authors call *constant data availability*.

Coda can be in one of three states (Figure 4.10). The whole file caching approach of AFS allows the client to cache in his local disk the files that he will need while disconnected. Caching files that are going to be needed in the future is called *hoarding*. Manual hoarding is possible but the authors have studied techniques to automate the task of selecting which files to cache. When the portable computer disconnects from the network, Coda is in the *emulation* phase: the user can work on the files cached in the local disk.

The servers containing replicas of a file form its *volume storage group* or VSG. Often, only part of the replicas are available (partitioning, disconnection), the *available VSG* or AVSG. Opening a file consists of reading it from one of the AVSG replicas and caching it locally. When the file is closed, it remains valid at the client, and a copy (reconciliation) is made to all AVSG servers.

Naturally, while disconnected the client is unable to receive any callbacks from the servers, and this presents an opportunity for conflicting updates on the same file. When the portable is later reconnected to the network, an automatic file system *reintegration* procedure is used, as illustrated in Figure 4.10. The procedure compares the versions of the client files with those of the server files and checks for conflicts. When no conflicts are found, the system automatically reconciles both versions of the file system. When conflicts are found, two versions of the conflicting files are stored and manual intervention of the user is requested.

4.3 DISTRIBUTED COMPUTING ENVIRONMENT (DCE)

In this part of the book we have referred to a number of technologies that help in the design and implementation of distributed applications and systems. Examples of these technologies are remote procedure call services, directory services, time services, distributed file systems, security systems, etc. For each

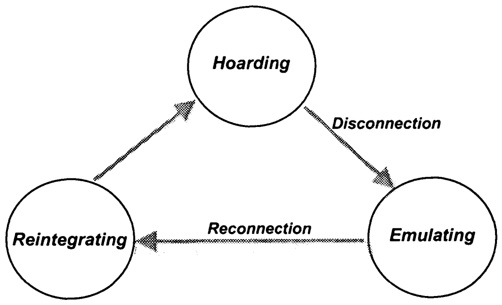


Figure 4.10. Coda Operation

of these services, several commercial products have emerged on the market. The diversity of technologies, often incompatible, makes the task of integrating applications made by different developers extremely hard.

The Distributed Computing Environment (DCE) is a standard endorsed by a consortium of several companies, including major players such as IBM, former DEC, and Hewlett Packard, under the name of Open System Foundation (OSF). DCE selects a particular technology for each of the services previously listed and offers them in an integrated package. The package was initially supported on Unix but was later ported to other operating systems as well.

Conceptually, there is not much really exciting in DCE, even though some of its services represent excellent technical solutions. This does not come as a surprise since DCE adopted standards and technologies that had already proven their value, some of which are addressed in this book. For instance, the directory service is based on X.500, the file system is derived from AFS, the time service is derived from NTP. The merit of DCE is to provide all these technologies as a coherent set.

Another feature that makes it hard to describe DCE in a few lines, is that it contains components that operate at different levels of abstraction. DCE is independent of the platform and operating system, thus these two layers are somehow outside of the scope of the DCE package. On top of the native operating system, DCE adds a dedicated threads interface. The availability of threads was felt to be a fundamental requirement for the efficient implementation of distributed applications (in particular servers), so DCE includes its own thread package. The *thread package*, like most of DCE components, has plenty of options and operational modes, enough to make almost everybody happy. For instance, three different scheduling policies are supported, priority based, round robin within the same priority and time-sliced round-robin; three types of mutexes are available; and so on.

Using the operating system (augmented with the DCE thread interface), the DCE *remote procedure call* package is implemented. The main computational model supported by DCE is client-server and RPC is a fundamental building block for the remaining services. Like almost every RPC package, DCE RPC allows server interfaces to be written in an Interface Definition Language and provides the compiler to automatically generate client and server stubs from these interfaces. Each service is identified by a unique identifier. To help programmers to obtain unique identifiers that are really unique, a unique identifier generator is also provided (which encodes the location and date of generation). The RPC package provides optional authentication and encryption (*see Secure Client-Server with RPC* in Chapter 18). On top of the DCE RPC service, the *time service*, the *directory service* and the *security* service are implemented.

The Distributed Time Service (DTS) is an evolution of NTP (*see Network Time Protocol* in Chapter 14). Its role is, of course, to keep local clocks synchronized. The service is of paramount importance for many other services. Among other applications, the global notion of time is used by the file system to timestamp updates and compare file versions. It is also used by the security service to check the validity of a credential. An interesting feature of the DCE time service is that the user is informed of the actual accuracy of the value provided. DTS uses this information when comparing two dates, to check that the timestamping error is small enough that they are comparable.

The directory service (names and structure are inspired by X.500, *see X.500* in this chapter) is organized as a set of cooperative *cells.* Each cell manages its own name space and has a local Cell Directory Server (CDS). To "glue" different cells, two global directory servers can be used: the DCE Global Directory Server or the Internet DNS. Cell directory servers interact with the global service to a Global Directory Agent, that shields the CDS from the details of the GDS or DNS.

The security server of DCE is based on the Kerberos *security server* (*see Kerberos* in Chapter 19). It manages access rights based on Access Control Lists and implements authenticated RPCs.

Finally, we can find the DCE file system, called the Distributed File System (DFS). It contains two main components: a local component, called *Episode*, and a global component based on AFS (*see AFS* in this chapter). In interesting feature of the DFS is that the file naming service is integrated with the directory service CDS, so files can be relocated just by updating directory data.

4.4 OBJECT-ORIENTED ENVIRONMENTS (CORBA)

In some sense, CORBA, the *Common Object Request Broker Architecture* is an object-oriented DCE. It has also been proposed by a consortium of major industry companies, the Object Management Group (OMG). Essentially, CORBA also follows a client server approach but at a higher level of abstraction. Instead of having client processes interacting with server processes, CORBA provides the basis for having objects interacting with other objects.

The state of a CORBA object is encapsulated by a well-defined interface. Like in an RPC system, object interfaces are written in an Interface Definition Language, in this case in CORBA IDL, whose grammar is a subset of C++. Following object-oriented principles, CORBA IDL supports inheritance, thus new interfaces can be built by extending previously defined interfaces.

OMG started by defining the architecture illustrated in Figure 4.11. The core of the architecture is the *Object Request Broker*, an abstraction that supports interaction among objects. The broker is responsible for making sure that an object can invoke another object, implementing the required protocols to send the requests and receive the replies. Of course, application programs need an interface to the broker in order to instantiate objects, create references to remote objects, issue object invocations and so on. The first CORBA 1.1 specification defined the CORBA IDL, the IDL mappings to common programming languages, and the application programming interfaces to the ORB. This allowed to develop application code that was more or less portable through ORBs from different vendors. The "catch" was that some vendors did include some non-standard features in their ORBs. These features were added to enhance the standard ORB functionality, but in practice, these proprietary enhancements prevented seamless portability. Another catch was that protocols and message formats were not part of the standard. The idea was to give room for each vendor to pick the most appropriate solution for their target market and architectures. The less positive aspect of that decision was that ORBs from different vendors did not inter-operate. This problem was eventually fixed with the release of CORBA 2.0, that defined a common protocol to be supported by every ORB, the Internet Inter-ORB Protocol, or simply IIOP (actually, to be more precise, IIOP is an implementation of a more *General Inter-ORB Protocol* (GIOP) over the TCP/IP protocol suite).

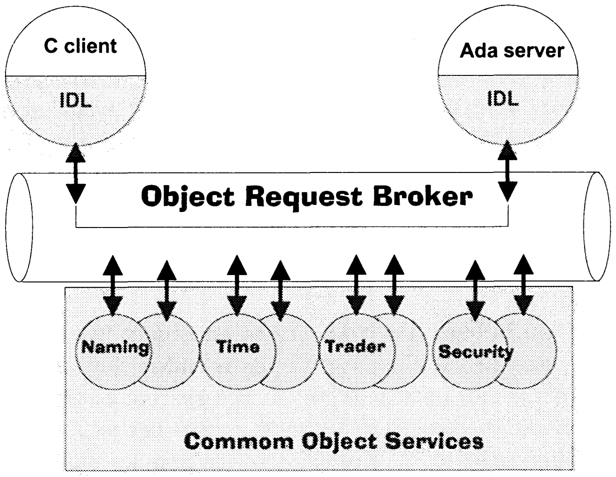


Figure 4.11. CORBA Architecture

If you have read the previous section on DCE, you already know that in order to build useful complex distributed applications you need more than remote invocations. CORBA has defined an extensive set of services, characterized by their standard CORBA IDL interfaces. No less than 15 services have been defined. We can describe some of the most relevant:

- *Naming Service:* as in any other system, it keeps associations among names and object references.
- *Persistence Service:* defines an interface to store the object state on *storage servers*, which can be implemented using traditional file systems or advanced database systems.
- *Concurrency Control Service:* provides a lock manager that can be used in the context of transactions to enforce concurrency control.
- *Transaction Service:* supports transactions, offering atomic commitment services.
- *Event Service:* allows some components to produce events that are distributed through an *event channel* to all interested *subscribers*.
- *Time Service:* provides a common time frame in the distributed system.
- *Trader Service:* allows objects to register their properties and clients to search for appropriate servers using this information.

Other services include the *Life Cycle Service*, the *Relationship Service*, the *Externalization Service*, the *Query Service*, the *Licensing Service*, and the *Collection Service.* In addition to these general purpose services, many interfaces have been standardized for specific business domains.

Of course, it is possible to build applications using just a few of these services. In fact, none of them is mandatory (but it is hard to build something useful without the naming service). The basic Corba functionality is pretty "conventional" when compared with an RPC system. The programmer writes the object interface in IDL. The IDL specification is compiled and a description of the interface is stored on the *Interface Repository.* From the IDL specification both client and server stubs are created for the target programming language (support for at least C++ and Java is now fairly common). The application programmer still has to write the actual object code, which is linked with the server stub and with an *Object Adapter.* The adapter supports the interface between the ORB and the object, providing the functionality to register the object within the ORB, to dispatch requests to the appropriate objects, and to send back replies.

The most straightforward way to activate an object is to execute it in the context of a dedicated process (this approach is called the *unshared server* approach). This process can be started when the system boots or just when an invocation is received by the ORB. As long as this process remains active, all invocations are forwarded to it. However, other policies can be implemented. For instance, it is possible to create a different process to execute each method. This approach, called the *server-per-method* approach is more suitable for stateless objects, where no shared state needs to be preserved on volatile memory across invocations.

Corba can be used to develop new applications from scratch. As with DCE, one advantage of using Corba is that many of the annoying details related with the implementation of RPC, server and client instantiation, etc, are handled by the ORB. An application built this way will be able to inter-operate with any other application adhering to the Corba standards. Additionally, "*transformer*" objects can be built to wrap legacy applications, adding Corba-compliant interfaces to old code. This type of architecture is known as a "Corba 3-tier client-server architecture" and is illustrated in Figure 4.12. Using this 3-tier architecture and Corba IIOP, it is also possible to build powerful applications for the World-Wide Web, but this is the topic of our next section.

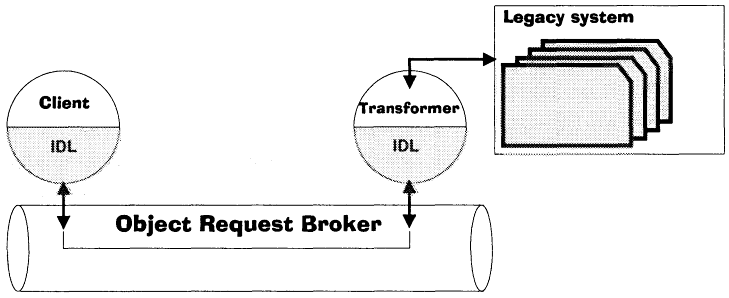


Figure 4.12. Corba 3-tier Architecture

4.5 WORLD-WIDE WEB

In the late 80's, despite the enormous potential and relative maturity of distributed systems technologies, it was felt among researchers that a "killer application" was lacking, one that could show the benefits of distributed systems in an obvious and indisputable way. The *World-Wide Web* (known simply by WWW or *the web*) was *the* killer application of the nineties.

It is interesting to observe that the application that had such a big impact in industry and society was in its inception relatively simple in terms of distributed systems concepts. It is basically a client-server application: clients, known as *browsers*, make requests to WWW servers in order to fetch documents and launch the execution of commands.

The WWW was created by Tim Berners-Lee and Robert Cailliau, working at CERN, with the goal of supporting information sharing among physics scientists. We can now say that the system was extremely successful in that task; it was the genesis of a global infrastructure that supports sharing and dissemination of information at a scale never seen before. The key for this success relies on the simplicity of its interface. Using a browser, remote information can be accessed just by clicking a button. Previously, cumbersome and often arcane sequences of commands had to be issued to achieve the same goal.

Documents in the WWW have a structured impure name called the *Uniform Resource Locator* (URL).
The URL has the following format: `<protocol>://<serveraddr>{/<path>}` where: the first field specifies which protocol must be used to interact with the server (several protocols are supported, being HTTP the most common); `serveraddr` is the address of the server to be contacted; and `path` indicates

which document should be fetched (if no name is specified, a document called `index.html` is read). Documents may be stored in several formats, from simple text to audio and video. Some of these formats are recognized and interpreted by the browser itself. Others are interpreted by companion applications that can launched by the browser in order to display the document.

Having a simple way to name and access documents is already a major contribution of the WWW architecture. However, if users were required to memorize the URL of all documents they were interested in, WWW would not have been such a big success. The other key factor of success was the use of *hypermedia* documents, which include *links* to other documents. The browser is able to interpret and display hypermedia documents in the *HyperText Markup Language* (HTML). Additionally, the browser allows the user to activate a link (typically, by clicking a mouse button) and automatically fetches the document whose URL is associated with the link. In this way, the user just has to remember the URL of the main page of an information repository or *site*, also known as the *home page*, which in turn holds the links for all other relevant documents (actually, most browsers have a way to store URLs, so that users do not even need to memorize the URLs of home pages). Today, it is a major business to provide pages, known as *portals* with links to useful information on the web, shops, advertising and much more. From a major portal, and just by clicking, the user is able to navigate through a huge net of documents, an often addicting activity also known as *surfing the web*.

The infrastructure we have just described is extremely useful and efficient, but it is somehow limited since it only supports the flow of information from the server to the client. Often, we also want the clients to send information to the server and request the execution of remote actions. For instance, the user may want to perform a query on a database, or issue an order when buying some goods. Thus, the first step to make the web more interactive was to allow clients to request the execution of programs in the servers. To support that type of interaction, servers were extended in several ways, the most common of which is an interface called the *Common Gateway Interface* (CGI). The CGI specifies how the browser indicates which programs should be executed and how parameters are sent to that program (and results sent back to the browser). According to this interface, WWW servers are able to launch programs upon request, which are executed as a separate process in the server machine. These programs can be binaries written in any programming language. However, CGIs are often interpreted programs written in popular script languages such as *perl*. CGIs can be fully-fledged applications or mere interfaces to other remote systems, such as database systems, forming a three-tier architecture as the one illustrated in Figure 4.13.

Typically, CGIs send results back to the clients in the form of HTML documents. Thus the CGI architecture allows to create web pages in run time. Pushed to the limit, this same concept allows to create a site where all the pages are created dynamically used information about format and contents stored in a database. The advantage of such system is that it simplifies the maintenance

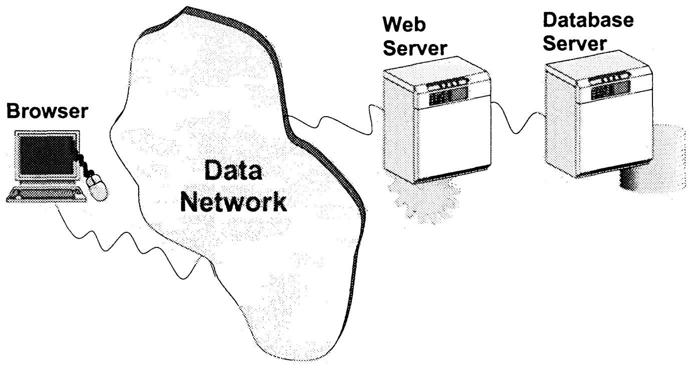


Figure 4.13. Three-tier WWW Architecture

and update of the pages, and allows pages to be customized according to the user profile.

Although the CGI concept is very versatile, it is not very efficient, because creating a new process is an expensive system operation. Often, the task performed by the CGI is extremely simple, for instance adding the time of day or the number of visitors to an otherwise static page. To avoid the need to start a new a process to execute these simple actions, most web servers offer extensions that allow some operations to be performed by the server itself, before sending the page to the client. The degree of complexity of the tasks performed by the server depends on the expressiveness of the language used to specify them. These extensions are known by several names, such as *Server Side Includes* (SSI), *Servlets* (which are Java applets executed by the server), or *active server pages* (Microsoft proprietary extensions). Some of these extensions support standard interfaces specially designed to support efficient access to database systems, such as Open DataBase Connectivity (ODBC) and Java DataBase Connectivity (JDBC)inxxJava DataBase Connectivity (JDBC).

We have just seen how to expand the processing capabilities of the server. We will now discuss how to expand the capabilities of the client browsers. It should be noted that today's browsers already have an impressive range of built-in facilities. However, it is impractical to include in the browser all the code to handle every possible format of multimedia documents. A way to make the browsers extensible is to allow new functionality to be attached dynamically to the browser. These extensions are known as *plugins.* Plugins have to be loaded onto the client machine and explicitly installed. To relieve the users from having to explicitly download and install plugins, the browser can be extended to accept code that is shipped together with a page.

One way to send code to the client is to send binaries, but this raises several problems. To start with, the server must have binaries for the different architectures where clients execute— an overwhelming task, given the myriad of architectures where browsers can run. This solution also poses problems to the clients— accepting and executing binaries from someone which may be not fully trusted raises severe security concerns.

An alternative its to allow the client to execute just a small number of commands, which are included in the page in the form of a script. Using this approach, the browser has a built-in interpreter of the scripting language. When parsing a page, the browser looks for scripts and executes them. Since the script can only execute the commands recognized by the browser, this solution is much more secure than accepting arbitrary binary code. The problem with scripting languages is that they are generally limited in the range of functionality they provide.

The *Java* language emerged has a excellent tradeoff between pure binary code and simpler scripting languages. By embedding a Java virtual machine in the browser, one can insert Java code in a web page. Being a very powerful language, Java supports the development of complex applications. On the other hand, the Java virtual machine ensures that the downloaded code does not violate safety or security constraints. For example, the browser can download smart graphic interfaces that serve as front-ends to other applications (that are executed in the server) or even download fully-fledged applications to be executed locally.

The combination of client side and server side processing turns the WWW into an environment where a wide range of distributed applications can be executed.

4.6 GROUPWARE SYSTEMS

The purpose of a groupware system is to support collaboration activities, including support for document sharing and exchange, in various formats and through diverse protocols. Users can be located in different physical locations and access data using different types of devices, including portable computers. A key feature of successful groupware systems is the *integration* of different data sharing mechanisms, like databases, electronic mail services, shared editors, replication managers, etc.

A groupware system often plays the role of a central information repository, a place where all the information required for a business process is held and can be manipulated. Typically, a groupware system also includes a workflow engine that keeps track of the document life-cycle. For instance, if a product order must first be validated by the section manager, the system may ensure that the order is not issued before being approved. Additionally, the system may also be configured to raise an exception if the order is not processed by the manager within 24 hours.

A notable groupware system is Lotus Notes. The major component of Notes is a hypertext document database that can store and link data coming from several sources including web pages, mail messages, images, etc. Users can query for documents using a full text search engine. Versioning capabilities allow users to update documents and keep track of changes. It is possible to exchange documents that contain links to other documents. For instance, a user can send a mail message to a set of colleagues and include a pointer to a working version of a document; by following that link, the colleagues will have access to the most updated version of the document.

Issues such as security, concurrency control, and replication management become particularly relevant in a context where sharing is the primary concern. Several security mechanisms are integrated in the Notes system, such as: mutual authentication between users and servers; powerful access control lists based on roles (like "author", "editor", "designer", among others); encryption of documents and communication channels; and the ability to sign documents.

Lotus Notes uses replication to place documents near their users. If an enterprise has several offices, it is possible to create replicas of a database in each site. Documents can be updated concurrently in different sites and the system supports automatic synchronization of replicas, using versioning control to detect which documents were changed and need to be copied. Clients can also replicate a portion of the database in their private machine, just to access data more efficiently or because they need to disconnect. Again, replica synchronization is performed when connectivity is regained.

An application programming interface allows using the basic Lotus mechanisms in the development of complex distributed applications. A set of development tools help programmers in this task. The tools include a dedicated scripting language (LotusScript), a formula language to build *filters* that can process documents, and design elements that ease the task of building graphical user-interfaces. As any other successful commercial product, Lotus also includes a myriad of interfaces to legacy and third-party database and systems, including, of course, the web.

Lotus can be seen as a good example of what is called *different-time-different-place* collaboration. This is probably the most frequent collaboration pattern. However, some other collaborative activities require a more tightly-coupled interaction, sometimes known as *same-time-different-place* or *synchronous interaction.* Tools for synchronous interaction include dissemination of audio and video, telepointers (a mechanisms where an user can point to a location in a document and the other users see the location pointed at), chat windows, shared drawing tools, etc. Such tools have the same sort of concurrency control and security requirements than the previous systems but exhibit much more stringent requirements in terms of connectivity.

4.7 SUMMARY AND FURTHER READING

This chapter gave examples of distributed systems and platforms. We started with the discussion of distributed name services and distributed file systems. Then, integrated platforms that include these and other services, such as DCE and CORBA, were presented. Finally, we briefly surveyed web and collaborative technologies.

More information on the implementation of DNS can be found in (Solomon et al., 1982; Bloom and Dunlap, 1986). An analysis of the DNS traffic is given in (Danzig et al., 1992). Alternative name server designs can be found in (Cheriton and Mann, 1989; Guy et al., 1990). A survey by Satyanarayanan on distributed file systems can be found in (Mullender, 1993).

There are several very complete books on DCE and CORBA. The book of (Shirley et al., 1994) provides a good introduction to the several DCE components. Interesting books on Corba are (Baker, 1997) and (Henning and Vinoski, 1999). A comparison between COM and Corba in given in citePitchard:99. Naturally, a large number of books about the Internet and the WWW are available, including the good introductions of (Comer, 1997; Abrams, 1998). For a short description of many WWW technologies, see (Spainhour and Quercia, 1996) and for a more detailed treatment see (Deitel and Deitel, 2000). Many books on Lotus notes exist, such as (Lamb and Lew, 1996) and (Haberman et al., 2000).

Table 4.2 gives a few pointers to information about some of the systems described in this chapter. Some of the sites are extremely complete repositories of distributed systems related software.

Table 4.2. Pointers to Information about Distributed Systems and Platforms

System Class	*System*	*Pointers*
Internet	**IETF**	www.ietf.org
	RIPE	www.ripe.net
	ISOC	info.isoc.org
	CIX	www.cix.org
	ICANN	www.icann.org
	IANA	www.iana.org
	Internet2	www.internet2.edu
	Internic	www.internic.net
	NGI	www.ngi.gov
	ISC	www.isc.org
Networking	**Cisco**	www.cisco.com
	Lucent	www.lucent.com
	***x*-Kernel**	www.cs.arizona.edu/xkernel/
	Triad	www-dsg.stanford.edu/triad/index.html
	Spinglass	www.cs.cornell.edu/Info/Projects/Spinglass/index.html
WWW	**WWW**	www.w3.org
	Apache	www.apache.com
	Netscape	www.netscape.com
	MS-Explorer	www.microsoft.com
Parallel Computing and Shared Memory	**MPI**	www-unix.mcs.anl.gov/mpi
	PVM	www.csm.ornl.gov/pvm/
	Parallel Tools	www.ptools.org/
	Top500	www.top500.org
	Spec	www.spec.org/
	ThreadMarks	www.cs.rice.edu/˜willy/TreadMarks/overview.html
	Alewife	cag-www.lcs.mit.edu/alewife
	Beowolf	dune.mcs.kent.edu/˜farrell/equip/beowolf
	Avalanche	www.cs.utah.edu/avalanche/
DCE	**Transarc**	www.transarc.com
Databases & Transacs.	**Open Group**	www.opengroup.org
	THOR	www.pmg.lcs.mit.edu/Thor.html
	TPC	www.tpc.org
	Arjuna	arjuna.ncl.ac.uk
Information Dissemin.	**Ninja**	ninja.cs.berkeley.edu
	Salamander	www.eecs.umich.edu/˜farnam/projects/collab.html
	Globe	www.cs.vu.nl/˜steen/globe
	W3Objects	arjuna.ncl.ac.uk/W3Objects/index.html
	Infospheres	www.infospheres.caltech.edu

Table 4.2 *(continued)*
Pointers to Information about Distributed Systems and Platforms

System Class	*System*	*Pointers*
ORBs and other Object Environms.	**OMG**	www.omg.org
	IONA	www.iona.com
	TAO	www.cs.wustl.edu/~schmidt/TAO.html
	Eternal	beta.ece.ucsb.edu/eternal/Eternal.html
	ORBacus	www.ooc.com
	Java IDL	www.javasoft.com/products/jdk/idl/index.html
	omniORB	www.uk.research.att.com/omniORB/omniORB.html
DCOM		www.microsoft.com
Multi-user Applics.	**DiamondPark**	www.merl.com/projects/dp/tour/index.html
	Spline	www.merl.com/projects/spline
	Sametime	www.lotus.com/home.nsf/welcome/sametime
	NetMeeting	www.microsoft.com
	MRObjects	www.cs.ualberta.ca/~graphics/mrobjects
	Maverik	aig.cs.man.ac.uk/systems/Maverik/index.html
Distrib. file Systems	**AFS**	www.angelfire.com/hi/plutonic/afs-faq.html
	NFS	www.ietf.org/rfc/rfc1813.txt
	Ficus	ficus-www.cs.ucla.edu/ficus
	Coda	www.coda.cs.cmu.edu
Distrib. O.S.	**Amoeba**	www.cs.vu.nl/pub/amoeba/
	QNX	www.qnx.com
	Mach	www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public
	Sprite	www.CS.Berkeley.EDU/projects/sprite
	EROS	www.eros-os.org
	PLAN 9	plan9.bell-labs.com/plan9dist
	GUIDE	www-bi.imag.fr/GUIDE/presguide.html
	Alpha	www.realtime-os.com/alpha.html
	Angel	www.soi.city.ac.uk/research/sarc/angel
	Inferno	www.vitanuova.com
	Grasshopper	www.gh.cs.su.oz.au/Grasshopper/index.html
Webcasting	**Marimba**	www.marimba.com
	Pointcast	www.pointcast.com
	TIBCO	www.tibco.com
	InfoBus	www.trl.ibm.co.jp/projects/ibr/index_e.htm
	iBus	www.softwired-inc.com/

5 CASE STUDY: VP'63– THE VINTAGEPORT'63 LARGE-SCALE INFORMATION SYSTEM

This chapter starts a case study that we carry throughout the book: The VP'63 (VintagePort'63) Large-Scale Information System. An imaginary wine company owning a traditional and obsolete information system starts a project aiming at its modernization. The case study is methodically addressed at the end of each part, so that we progressively improve VP'63. In this part, we start by making it: modular, distributed and interactive.

5.1 INTRODUCTION

We start our running case study, which we will develop throughout the book. At the end of each part, we apply the concepts addressed in that part. The purpose of the case study is to exercise the skills of the architect in developing an architecture, and ultimately assessing how well the notions discussed in the book were assimilated. In consequence, we will use the style of a dialogue inside a team of system architects, and will intentionally not define or refer to the places where the terms and concepts used have been previously treated in the book.

An imaginary traditional Portuguese wine company owns an obsolete information system. The company management has devised an ambitious strategy for enhancement of the system in support of current and emerging business, and has contracted a team to develop an architectural project for that development. The corporate strategy makers traced the following objectives:

- Seamless business information support system, from shop floor, through offices, to higher management, enabling applications to give coherent and up-to-date information about the state of the business.
- Coherent document management support system, allowing the design of simple applications that reliably disseminate persistent information created by several producers to the whole or groups of enterprise collaborators.
- One-PC-per-employee strategy, adapted to the real circumstances (e.g., rural workers) ie., at least one-PC-per-employee-group.
- Improved automation of the wine processing facilities, aimed at a better quality of the core process (wine making) whilst retaining the traditional ways, and at more effective handling of the ancillary processes (bottling, corking, labeling, etc.).
- Integrated industrial management support system, allowing the design and/or installation of multiparty interoperative applications, a prompt, real-time perception of the shop-floor state from several places in the company, and an integration with the business information system.
- On-line transactions in two facets: a Web portal oriented towards the wine culture, featuring historical and informative contents and a virtual wine shop supporting direct customer-to-business commerce; and a virtual enterprise network connecting the company to its suppliers and downstream wholesale clients, supporting business-to-business commerce.

The project received the code name of *VintagePort'63 Large-Scale Information System*, VP'63 in short, and the team will be composed by the authors and the reader. The case study is methodically addressed at the end of each part, so that we progressively improve VP'63. Of course, this strict sequence relates to the book structure. In a real project the architect had better tackle all the facets concurrently— distribution, fault tolerance, real-time, security, management— in a spiral of development that clarifies their interdependencies and eliminates conflicting objectives, until the final architecture.

5.2 INITIAL SYSTEM AND FIRST STEPS

The company has several vineyards in the country, with local offices and processing plants in some of them. The central offices are in Porto, the capital of the famous Port Wine. The information system, like many others of the earlier generations, is mainframe-based, centralized, without Internet access. Remote facilities access it through remote login, via virtual terminals on PCs, connected to the mainframe through leased lines, as exemplified in Figure 5.1a.

More recently, some of the larger offices, co-located with the processing plants, have augmented their computing needs, such that ad-hoc solutions were put in place. This mostly consisted of installing mid-range servers with local databases and hosting local support services, both office and production management. However, this evolution created a potential for inconsistency with the mainframe system, which in principle must hold a coherent state of the business. In fact, this situation can be described as having semi-autonomous subsystems or *islands*, now detailed in Figure 5.1b. These islands must perform periodical explicit state reconciliations with the central mainframe system. It had been decided that these operations take place during the night, every day, since they involve stopping the system, making a series of file transfers, and running consolidating transactions.

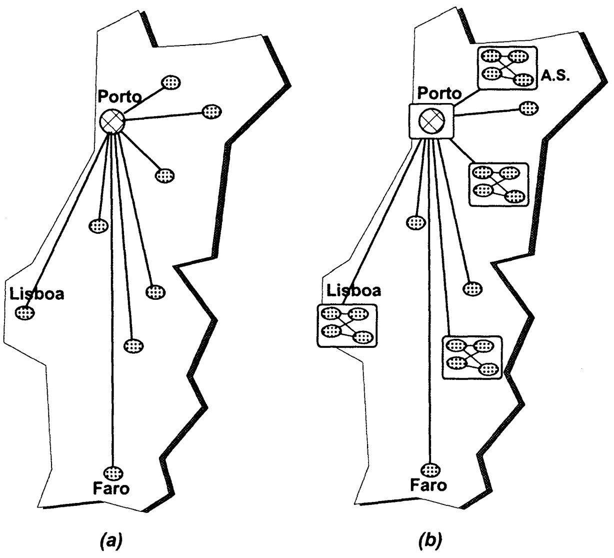


Figure 5.1. (a) Centralized Mainframe (rlogin); (b) Autonomous Subsystems/Islands (ftp)

5.3 DISTRIBUTED COMPUTING APPROACHES

In what concerned distribution, the team agreed that the strategy for the system evolution should be traced along the following lines:

- maintaining centralized business control, whilst allowing the deployment of distributed services;
- achieving openess, through the use of COTS systems (e.g., Linux and WNT), protocols (e.g., TCP/IP, HTTP), and infrastructures (e.g., Internet);
- distributing processing activities, for modularity in face of fast changing business configurations;
- distribution of data repositories for information and resource sharing;
- enhancement with proprietary middleware when applicable;
- distribution should have in mind availability and performance enhancement, to be addressed in later stages.

***Q.1. 1** What should be the evolution in terms of networking infrastructure?*

The networking setting of the company migrated to the Internet. The autonomous islands were already networked internally through local area networks running TCP/IP, that connected via the leased lines to a special purpose gateway from TCP/IP to the mainframe's native communication architecture. In consequence, all islands have been fitted with routers connected to an ISP via an adequate link in terms of speed and throughput.

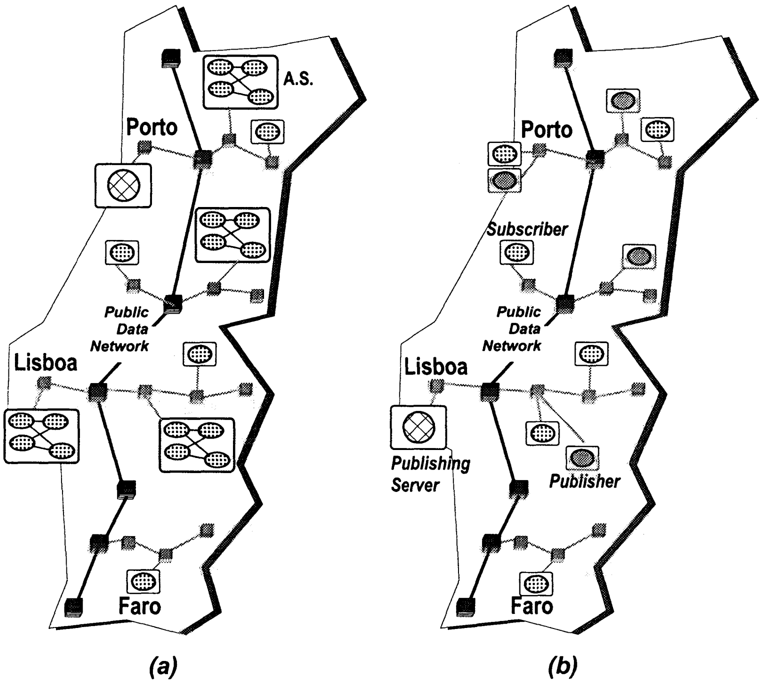


Figure 5.2. (a) Client-Server; (b) Publisher-Subscriber

On the other hand, the PCs hosting the isolated remote terminals were converted to fully working client units capable of performing local computing, and connecting through the Internet to local services (to be defined), services on the main system, and other local units. The connection was designed using a small ISDN dial-up router that allows a small-scale expansion of the number of remote clients at a facility. The new networking layout can already be perceived in Figure 5.2a. The application of the WAN-of-LANs principle was deferred to a later phase, when the installation of a secure VPN will be considered. After this phase there will be a desirable homogeneity of the architecture amongst what are now islands and isolated remote client positions. This will be beneficial both in terms of architectural coherence and modularity, and potential for reconfiguration of the company's information system layout. This was a requirement from the corporate strategy makers.

Q.1. 2 What are the adequate paradigms in what concerns the organization of distributed activities and services in such a company?

The team reflected on the business objectives present and future, traced for the company by the corporate strategy makers. Two paradigms were seen as enabling the installation of applications serving the outlined strategy:

- Client-server, enabling: access of remote facilities to central services; access to services in the autonomous subsystems by the local clients; transactional access of Web clients, both internal (employees) and external (e-commerce clients).
- Publisher-subscriber, enabling: event-based handling of generic management information in a content-sensitive manner; event-based handling of time sensitive production information, for seamless integration between the production management and the general management systems.

Q.1. 3 How should client-server and publisher-subscriber subsystems be set up?

Figure 5.2a illustrates the client-server set-up for the whole company. The main database remains at the headquarters in Porto, but the database engine is provided with a transactional front-end supporting remote queries and updates from the clients residing at the several company facilities in the country. The operations are essentially the same as supported previously by the closed mainframe. However, there is a potential for adapting old applications and writing new applications to take advantage of local client processing power, instead of loading eveything onto the mainframe. Furthermore, this opens the way to 3-tier computing from thin Web clients at a later stage.

Figure 5.2b exemplifies how a publisher-subscriber subsystem should be set up. This concerns support for applications handling the part of the documental information that requires a push treatment (billboards, memos, and any changes requiring prompt attention in regulations, price lists, production information such as stock/orders, etc.). In consequence, the architecture must allow for several publishers that may be in different facilities. The publishing engine resides in Lisboa, where the company has important administrative staff offices with a powerful server, which will be adapted to be the publishing engine. This is a persistent repository which holds the publication rules for the subscribers. For example, new price lists are disseminated to commercial department heads, memos are disseminated through the relevant subscription list, finished production batches are made available as new stock, etc. At least part of this information can also be made available through front-ends such as news readers.

5.4 DISTRIBUTION OF DATA REPOSITORIES

The former set-up is still based on a central mainframe database. Given the increased dependency on the information system, situations of bottleneck on the central database may arrive. Besides, parts of the central database were already shipped every night to the autonomous islands, since their operation was too intensive to support remote accesses for every operation. This creates a daily inconsistency that twists the business-centric computational model applications are supposed to comply with, by definition of the enterprise model. The team identified a number of potential problems deriving from the analysis above:

- peak situations will become frequent when the main database in Porto will be overloaded;
- when the main database server in Porto or the connection to it fail, the whole enterprise operation stops;
- situations of network partitioning in long-haul connections from distant facilities are more probable;
- there is a potential for conflicting operations over the day between different autonomous islands, and between the latter and remote clients.

The increased informatics content of the company's information flow shrinks reaction times and increases the frequency of transactions. In consequence, this situation may assume a dramatic proportion if nothing is done to address it.

Q.1. 4 How can the performance, availability and consistency problems created by a centralized database and ad-hoc caches be solved?

All this points to the distribution of the information repository, by means of a distributed database management system. Fragmentation of the central database and its distribution by several of the main facilities of the company is a mandatory step, depicted in Figure 5.3a. This is easy to do since all main facilites were planned to have high quality access to the Internet. Criteria for fragmentation should pay attention to data importance, functionality and locality. Criteria for distribution should match the criteria for fragmentation, placing fragments with local information in the relevant areas, functional fragments with the corresponding department locations, critical fragments in highly-protected and/or highly-accessible locations, depending on the perspective of criticality being integrity or availability.

Fragments whose definition deserves special attention are those of the autonomous islands. For performance reasons, information accessed often should be near each location. Instead of this being done through loosely consistent caches, the persistent information repositories of the islands should be redefined as fragments of the main database, as illustrated in Figure 5.3b. The team devoted special attention to the definition of these fragments. If properly designed, most of the transactions on an island will exhibit the property of locality, not burdening the main system. A refinement was considered important though: part of the information resident in the islands databases was read-only information copied from the database and shipped to and cached in all islands on a daily basis. That information cannot stay in a single fragment, except the one at the main database site. The team identified two solutions: (a) in anticipation to the reformulation of the information flow, part of this information will circulate through the information dissemination subsystems, i.e., publisher-subscriber and distributed Web-based file system (to be defined); (b) in anticipation to the measures to achieve fault-tolerance, where database replication will be foreseen, at this stage through read-only replicas for performance reasons.

that the database is depicted whole at the headquarters location in Figure 5.3b, despite being fragmented elsewhere.

It was also decided to keep a copy of the complete database at the headquarters location, as depicted in Figure 5.3b. The consolidated copy is achieved by periodically reconciling all the external fragments. This was planned in order to facilitate the transparent operation of strategic packages such as data warehousing, data mining or executive information systems, which require a full image of the business information system. Some of them are resource and power hungry, and making them operate on a distributed database would have a negative effect on performance. This way they can operate on a central database image without disturbing the performance of the rest of the system.

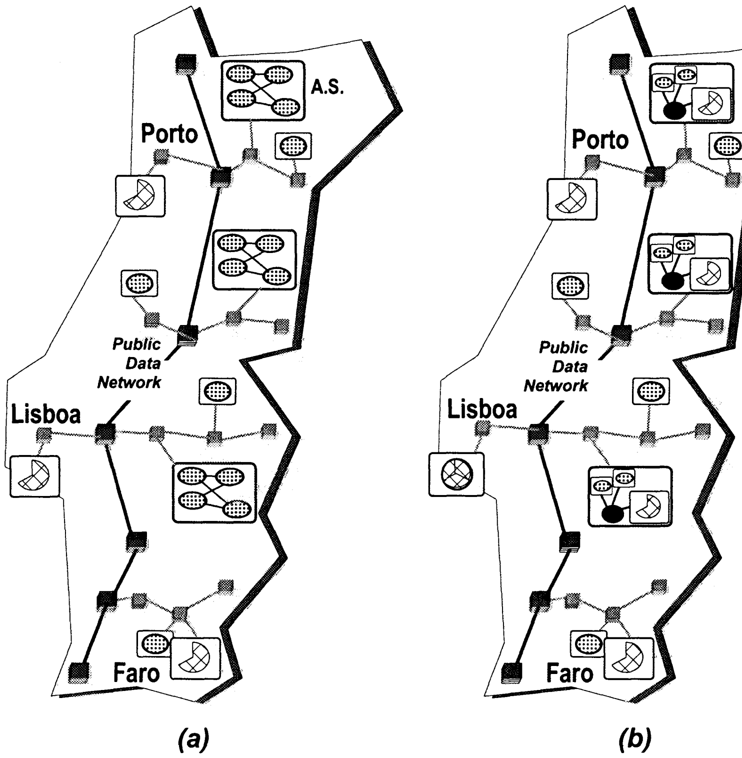


Figure 5.3. (a) Fragmentation of Central DB; (b) Consolidation of the Island's DB Fragments

5.5 DISTRIBUTED FILE SYSTEM ACCESS

Amongst the distributed access to services, the file service access is of paramount importance. It enables the transparent distribution of file-based applications and is in itself a reliable way of disseminating information that does not change too often.

***Q.1. 5** What kind of distributed file service is adequate for a geographically large-scale setting such as VP'63?*

The team specified a large-scale distributed file system (DFS) architecture with facilities for setting-up read-write (RW) and read-only (RO) files or volumes, distributed by several servers. The file system model, exemplified in Figure 5.4a, should be of the upload-download type, with server files cached in local client disks. This overcomes the delay and instability of WAN communication. Transactional file access to both RO and RW volumes makes it easy for human or computer clients to use the system as a file-based information dissemination/archival infrastructure, which can be combined with the event-based publisher-subscriber mechanisms already described. Replicated RO volumes support long-term publishing: a single writer sporadically modifies files, and releases them onto all RO copies at the distributed company sites (e.g., billboards, general regulations). Alternatively, shared RW volumes support moderately frequent single- or multiple-writer updates (e.g., global company phone directory, multiple source FAQs, or price lists, etc.). All of them should be considered as a logical part of the global information system, as suggested by the way the main file server is depicted.

Human users, mainly non computer-literate users, should be given access through the Web in as many situations as possible, given its simplicity. In addition to the existing 3-tier solutions for database access currently provided by all DB-engine manufacturers, the team studied the enhancement of the file access to provide 3-tier Web-based access to the large-scale DFS. This set-up, depicted in Figure 5.4b, has an enormous scability, but deceiving simplicity: the core infrastructure of file system servers fuels the DFS client's caches located strategically in the company infrastructure, as the second level of the hierarchy. Both servers and caches are the local file systems on which HTTP servers run, serving pages to the third level of the hierarchy, the HTTP client browsers. The set-up implies that contents to be disseminated and later browsed from, be edited in the desired language/format: ascii, html, scripting languages, etc.

Further Issues

The project needs now some refinement, and the reader was assigned the study of a few questions that were still left to be solved:

***Q.1. 6** What routing policies should be used in the application of the WAN-of-LANs principle to the networking infrastructure of VP'63?*

***Q.1. 7** What kind of protocol architecture/stack should be used for enabling client-server RPC access to the database server?*

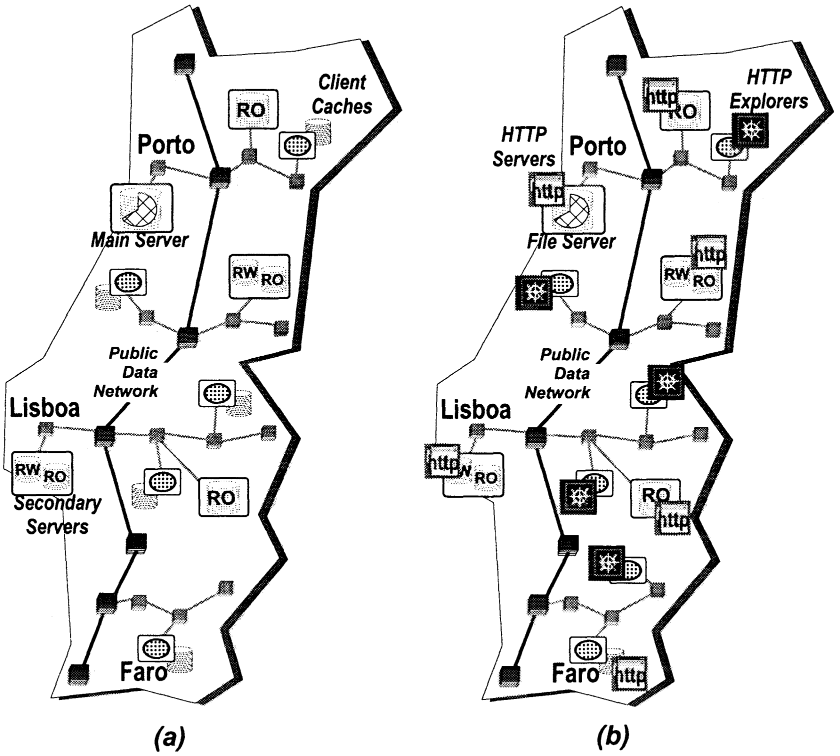


Figure 5.4. (a) Distributed File System; (b) Web-based File Access

Q.1. 8 Design the detailed architecture of the publisher-subscriber subsystem using group communication.

Q.1. 9 Define criteria for fragmentation of the database and location of the relevant fragments, based on a generic classification of information criticality and locality.

Q.1. 10 Given that it was once possible to have the autonomous systems working for a full day completely disconnected, will it be possible to configure the system so that they are connected to the central system only a few times a day, reducing the networking costs? Determine a policy for that and its implications on consistency.

Q.1. 11 The same question above applies to the DFS. Define a policy for use of a weakly consistent file system that supports disconnected operation, and its implications on consistency.

Q.1. 12 Consider the transactional DFS studied: (a) would it easily support a low read/write ratio with single writer?; (b) would it get worse with highly *competitive multiple writers with high read/write ratios?; (c) and with low read/write ratios?*

Q.1. 13 *Design the detailed architecture of the 3-tier DFS-Web infrastructure, including layout and update mechanisms. Suppose that the core DFS service is hosted in three main servers. Optimize both the freshness of information updates and the response time of Web page requests from end clients.*

II Fault Tolerance

Why do computers stop and what can be done about it?
— Jim Gray, 1986

Contents

Overview

Part II, Fault tolerance, addresses dependability of distributed systems, that is, how to ensure that they keep running correctly. It contains the fundamental notions concerning dependability, such as the triad fault-error-failure and provides a comprehensive treatment of distributed fault tolerance. Chapter 6, Fundamental Concepts of Fault tolerance, starts with the generic notion of dependability and its associated concepts, and ends with the introduction of distributed fault tolerance. In fact, distribution and fault tolerance go hand in hand, since the former requires the latter to keep reliability at an acceptable level, and the latter is made easier by some qualities of the former, such as independence of failure of individual machines. Chapter 7, Paradigms for Distributed Fault Tolerance, discusses the main paradigms of this discipline. After introductory concepts and notions about fault-tolerant communication, it addresses issues such as: replication management, resilience and voting, and recovery. Chapters 8 and 9, Models of Distributed Fault-Tolerant Computing and Dependable Systems and Platforms, show how to incorporate fault tolerance in distributed systems. Explaining the main strategies for the diverse fault models, its materialization is discussed for remote operation, diffusion and transactional computing models. Finally, concrete system examples are given. Chapter 10 continues the case study: making the VP'63 System dependable.

6 FAULT-TOLERANT SYSTEMS FOUNDATIONS

This chapter addresses the fundamental concepts concerning fault tolerance. It starts by introducing the notion of dependability and discussing why it is difficult to build dependable systems. The evolution of fault-tolerant computing is reviewed, from hardware fault tolerance to distributed software-based fault tolerance. Finally, the chapter introduces the most relevant architectures for fault-tolerant communication and processing, that are later described in detail in the subsequent chapters of this part.

6.1 A DEFINITION OF DEPENDABILITY

Compared with simple but nevertheless useful instruments (such as a hammer, for instance) computer systems seem to be rather fragile artifacts: they often do not behave as we expect them to, and usually decide to do it at the most inconvenient moment. This undesired behavior has two main causes. The first is that computers are complex systems, made of many different hardware and software components. These components interact with each other in ways often not predicted by the system designer. It is a challenging task to create the appropriate architectural constructs to address the mismatches caused by this complexity (the hammer always seems to work fine, even when everything starts looking like a nail). The second reason stems from an old rule of engineering, known as Murphy's Law. Put simply, this law states that if we neglect the possibility of hazards occurring, they tend to occur in the worst possible manner at the worst possible moment. This chapter presents the fundamental notions required to build dependable computing systems, i.e., computing systems that behave like their users expect.

We say that a system is *dependable* if it exhibits a high probability of behaving according to its specification. This rather simple statement hides a number of subtle issues. To start with, it assumes that a comprehensive specification of the system behavior is available. This requirement is sometimes overlooked: it is not simple to derive a complete and unambiguous specification of the system from user requirements that are often fuzzy or implicit. Additionally, a complete specification should not be limited to what the system does but must also specify the environmental conditions required for the system to provide the desired service. Most people realize that a personal computer is not water-proof but few people realize what exactly happens when the computer is exposed to high temperature or heavy dust.

Another ambiguous issue in our definition of dependable system is the notion of high probability. How high is high? This naturally depends on the purpose of the target computer system: the requirements of a life-supporting system and of a video-game console are quite different. The consequences of a failure are much more dramatic in life-supporting systems than in a gaming machine (even though the resistance of an arcade console to physical damage needs to be probably higher than that of a medical instrument). The knowledge of the required degree of dependability is important because, as you may expect, dependability does not come for free. Paraphrasing Laprie (Laprie, 1992), dependability is then:

Dependability - the measure in which reliance can justifiably be placed on the service delivered by a system

Is there a systematic way to achieve such reliance justifiably? To start with, we must understand what are the *impairments* to dependability, i.e., the potential causes for incorrect behavior. Then, we must learn about the *means* to achieve dependability, i.e., the techniques that allow us to achieve correct behavior despite the impairments. Finally, we must devise a way to express the level of dependability desired and assess whether it was achieved, by defining dependability *attributes*. Each of these issues will now be addressed in turn.

6.1.1 Fault, Error and Failure

The impairments to dependability assume three facets: fault, error, and failure. When the system behavior violates its service specification we say that a *failure* occurs. Building dependable systems is about preventing failures from occurring. However, to be successful, we must understand the process that leads to failure, which starts with an internal or external cause, called *fault*. The fault may remain *dormant* for a while, until it is activated. For example, a defect in a file system disk record is a fault that may remain unnoticed until the record is read. The corrupted record is an *error* in the state of the system that will lead to the failure of the file service when the disk is read. The failure is thus the externally observable effect of the error. It should be noted that similar failures can be derived from quite different errors. A screen filled with strange characters can be the visible result of either a defective video card or a flawed operating system routine. On the other hand, errors are sometimes not immediately visible at the system interface. In the disk example, a long time may elapse before that particular record is read. Such errors are in the *latent* state until they are detected and/or they produce a failure. As with failures, the same error can be caused by different faults. For instance, the disk error may be due to a physical fault in the disk surface (bad record), but it may also be due to a misalignment of the disk head.

There is a wealth of fault types, which can be classified along several axes or viewpoints. There is a fundamental distinction of the phenomenological origin of faults: *physical*, generated by physical (hardware) causes; *design*, when introduced during the design phase; *interaction*, when occurring at the interfaces between system components, or at the interfaces with the outside world. Design faults, and some interaction faults, are caused by humans. Faults may also be classified according to the nature (accidental or intentional, malicious or not), the phase of creation in the system's life (development or operation), the locus (internal or external), the persistence (permanent or temporary). A classification is proposed in (Laprie, 1992). Most relevant to distributed systems are interaction faults, since they target interactions between distributed components. Amongst them, temporary faults assume special relevance: *transient* (external) faults mainly affect communication (for instance, electromagnetic noise due to a spark); *intermittent* (internal) faults mainly affect concurrent programs, so typical of distributed systems (for example, races and deadlocks).

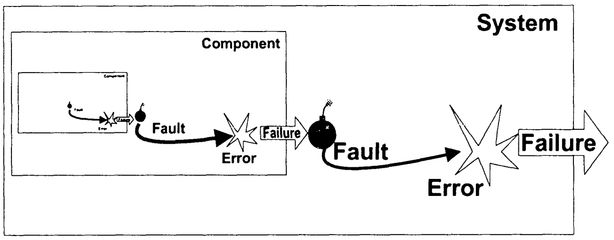


Figure 6.1. Fault, Error and Failure

The fault, error, and failure definitions can be applied recursively when a system is decomposed into several components:

fault → error → failure → fault → error → failure ...

That is, an error inside the system is often caused by the failure of one of its components, which at system level should be seen as a fault. Figure 6.1 illustrates this recursion. Although we should avoid ambiguity when addressing the mechanisms of failure, it is thus possible to address the same phenomenon as a (component) failure or a (system) fault, depending on the viewpoint. Likewise, interactions between components may fail in several ways (e.g., timing failure) constituting system-level faults that lead to an erroneous state (e.g., timing error). Faults may cause other faults. An error may give rise to other errors, by propagation, and in fact, a failure may be at the end of a chain of propagated errors.

6.1.2 *Achieving Dependability*

As we have seen, there is usually a cause-effect chain from a fault to a failure. To achieve dependability one should break this chain by applying methods that act at any point in the chain to prevent the failure from occurring. The source of the chain, the faults, are thus the natural targets of several means to achieve dependability. These means can be used in isolation or, preferably, in combination.

The first approach we study is called *fault removal.* It consists of detecting faults and removing them before they have the chance of causing an error. Targets of fault-removal include software bugs, defective hardware, and so forth. *Fault forecasting* is the set of methods aiming at the estimation of the probability of faults occurring, or remaining in the system. Some classes of faults are easier to detect and remove than others. In consequence, fault forecasting can be seen as complementing fault removal, by predicting the amount of residual faults in the system.

Fault prevention, as its name implies, consists of preventing the causes of errors by eliminating the conditions that make fault occurrence probable during system operation. For instance, using high quality components, components with internal redundancy, rigorous design techniques, etc. The combination of fault prevention and removal is sometimes called *fault avoidance*, i.e., aiming at a fault-free system.

Of course, not all faults can be prevented from occurring during system operation, whereas other faults may even be present from the beginning of operation, having eluded fault removal. In consequence, one must create complementary mechanisms that block the effect of the fault before it generates a failure. In such case, we say that the system is capable of providing correct service despite the occurrence of one or more faults or, in other words, that the system is *fault-tolerant* (FT). Fault tolerance acts at the stage of error production, through mechanisms that are designated by *error processing*. Upon identification of an error-producing fault, it is desirable to eliminate it, as soon as possible, in what is called *fault treatment*.

6.1.3 *Measuring and Validating Dependability*

Fault avoidance and fault tolerance are strategies that can help the system architect to build dependable systems. But how to assess the degree of success of these strategies, i.e., how to *measure* and *validate* the degree of dependability attained by a system? This is expressed through the following *attributes*:

Reliability	the measure of the continuous delivery of correct service
Maintainability	the measure of the time to restoration of correct service
Availability	the measure of the delivery of correct service with respect to the alternation between correct and incorrect service
Safety	the degree to which a system, upon failing, does so in a non-catastrophic manner

Recall that we have attached a probability to the notion of dependability as provision of correct service. Several of these metrics of dependability can be expressed as probabilistic functions. *Reliability* can be equated with the probability that the system does not fail during the period of mission of the system (e.g., a flight). For continuous mission systems (e.g., web servers), reliability can also be expressed by the *mean time to failure* (MTTF) or by the *mean time between failures* (MTBF). Finally, reliability can be expressed as a failure rate probability, for example, 10^{-9} *failures per hour*, a typical figure for safety-critical systems. Another metric of dependability is *availability*, the probability of the system being operational at any given time, when it alternates with failed states. *Maintainability* is the attribute defining the time needed for the system to recover from a failure. Given reliability MTBF and maintainability MTTR (mean time to repair) of a system, availability can be expressed as *MTBF/(MTBF+MTTR)*. For the sake of example, Table 6.1 shows the corresponding downtime per year for several availability figures. Worthwhile noting is the capability of defining a continuum between full service and complete interruption of service, expressed as *performability* (Meyer, 1992). This attribute quantifies the capacity of graceful degradation of a system. In other words, it offers a combined metrics of performance and dependability, which can be seen as a 'dependability' view of *quality of service* (*see Quality-of-Service Models* in Chapter 13). Another important attribute is *safety*, which is equated with the conditional probability of, given a failure, it not being catastrophic. Other attributes that can be considered of dependability are those concerning *security*, such as the preservation of confidentiality or integrity. Security is discussed in Part III of this book.

Given a specification in terms of system dependability attributes, it is important to *validate* whether the latter are attained. The distinction is often made between *building the right system*— validation in general terms— and *building the system rightly*— verification of the design and implementation— as two facets of this process. This couple is often referred to as *Validation & Verification*, or simply *V & V* (Boehm, 1988). One cannot simply put the system into operation and measure how often it fails; usually the desired probability of failure is so low that this approach is infeasible. In the early design phases, the functional and design specifications of the system should be subjected to *design validation*, with the aim of assessing whether the proposed system satisfies the requirements. Later, the implementation specifications (e.g., algorithms and protocols) and the implementation itself (e.g., code) should be subjected to

Table 6.1. Downtime per Year for Several Availability Figures

Availability	Down-time/year	Example Component
90%	$>$ 1 month	Unattended PC
99%	$\simeq$ 4 days	Maintained PC
99.9%	$\simeq$ 9 hours	Cluster
99.99%	$\simeq$ 1 hour	Multicomputer
99.999%	$\simeq$ 5 minutes	Embedded System (PC tech.)
99.9999%	$\simeq$ 30 seconds	Embedded System (special HW)
99.99999%	$\simeq$ 3 seconds	Embedded System (special HW)

implementation validation, to check whether the system is correctly built. For instance, an automatic tool can check a specification (made in a formal language), against system properties (also formally specified): this is called *formal verification*. This sort of verification can also be made in implementation code, by tools that perform exhaustive code walks. Another important class of validation techniques is *dependability evaluation*: techniques in this class allow to quantify the dependability of a system based on the dependability of its components. Examples of such validation techniques commonly used in dependability work are *fault-injection* and *software reliability* evaluation. Fault-injection, as the name implies, consists in artificially generating faults in a target system and observing the resulting behavior. Software reliability evaluation can be done using techniques such as statistical trend analysis.

In conclusion, two relevant techniques to achieve dependability are fault removal and fault forecasting which complement each other. The resulting architecture can then be subject to dependability validation to assess in what extend the dependability attributes are met. Table 6.2 summarizes the impairments, means and attributes of dependability that we have just discussed.

Table 6.2. Main Dependability-Related Concepts

Impairments	Means	Attributes
Faults	Fault Removal	Reliability
Errors	Fault Forecasting	Maintainability
Failures	Fault Prevention	Availability
	Fault Tolerance	Safety

6.1.4 Fault Assumptions and Coverage

In order to avoid or tolerate faults we need to understand how, how often, and for how long they occur. Thus, the first step to building a fault-tolerant system is to define the *fault model*, i.e., the number and classes of faults that need to be tolerated. Obviously, it is possible to classify faults according to many different criteria. In this book we are mainly concerned with system faults in *interactions* between components, or in other words, faults concerning actions whose result is observable outside the component, since these are the most relevant in distributed architectures (network messages, input-output observations and actuation, clock readings, disk reads and writes, etc.).

The most benign classes of faults belong to the *omissive* fault group. These faults are characterized by the component not performing some interaction when specified to. *Crash* faults occur when a given component permanently stops operating. *Omission* faults occur when a given component omits an action from time to time. *Timing* faults occur when a component is late performing an action. The delay between the specified instant for the action to take place and the actual instant when the action is observed is called the *lateness degree.* Note that an omission fault can be seen as a particular case of timing fault, that exhibits an infinite lateness degree (the action never takes place). Likewise, if *omission degree* is the number of successive omission faults, then a crash fault is a particular case of omission fault, with an infinite omission degree (all actions after a certain point are omitted).

The *assertive* fault group is characterized by the component performing some interaction in a manner not specified. Assertive faults are further divided into *syntactic* faults, when the construction of the interaction is incorrect, and *semantic* faults, when the meaning conveyed by the interaction is incorrect. A syntactic fault is a semantic fault where the construction is also incorrect. Consider a room temperature sensor interacting with a controller. The output of the sensor is defined to be a signal sign (+ or−) followed by two numeric digits. Readings such as "+ab" or "*24" can be marked as erroneous by a syntactic analyzer (an error detector in fact) while a reading of "-99" can only be detected as incorrect by using the application semantics (for instance, if you know that the target can *never* reach that temperature) or through comparison with redundant information (held by the user, or from other correct components).

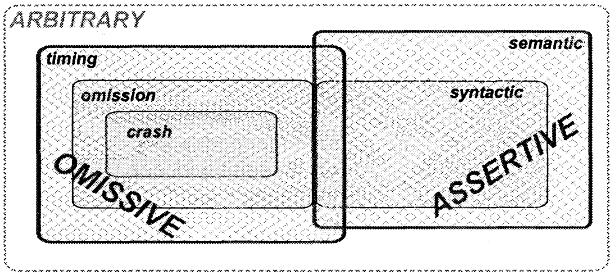


Figure 6.2. Classes of Interaction Faults

Figure 6.2 illustrates the relationship between the fault classes just described. When interactions are multi-component (e.g., a multicast transmission), faults may affect all concerned components, in which case they are said to be *consistent*. Otherwise, they are *inconsistent* (e.g., omission faults at just some of the recipients).

Omissive faults occur essentially in the time domain. Assertive faults occur in the value domain. Inconsistency introduces the space dimension. When these dimensions can combine, we have *arbitrary* faults. The arbitrary fault class is used when one does not wish or cannot make any assumptions on the behavior of faulty components. Obviously, this should be understood in the context of the universe of "possible" faults of the concerned operation mode of the component. We recall that we are interested in interaction faults. Practical systems based on this baseline assumption normally specify quantitative bounds on the number of faults, or at least quantify the tradeoffs between the resilience of their solutions and the number of faults eventually produced (Babaoğlu, 1987).

Of course, this behavior seems overly pessimistic, and needless to say, expensive to tolerate. An arbitrary fault can be caused by an improbable but possible sequence of accidental events, which for example may lead to a catastrophic failure of a safety-critical system. It can also be caused by the meticulous action of an intentional and malicious component (an intruder) that deliberately tries to defeat the system protections. Both cases (safety and security) may justify the adoption of such a restrictive fault model. A well-known subset of arbitrary faults are the *Byzantine* faults after a paper by (Lamport et al., 1982), that denotes inconsistent semantic faults (e.g., sending different messages to different recipients). A particular case of arbitrary fault occurs in an interaction that takes place before being expected. Although it might be called an early timing fault, it is akin to a forged interaction (Powell, 1992). In this book, timing faults always refer to (omissive) late timing faults.

System interaction faults are immaterial: they take place in the context of protocols running between components. In fact, they are very appropriately *component failures*, in the viewpoint of the components suffering them. Throughout this book, we will use either term— system fault or component failure— depending on the viewpoint. For example, we will talk about an omission or timing failure in a communication network component, which leads to an erroneous state of the communication system that we generically call an omission or timing error, and may discuss how to make protocols timing or omission fault-tolerant.

A system tolerating two arbitrary faults is not necessarily better than one tolerating two omissive faults, and this one not necessarily better than a third system tolerating one omissive fault. Recall that our purpose is to minimize the number and the probability of occurrence of failures. If in worst case, the probability of the system doing more than a single omissive fault was negligible, then all the three systems would be just as good. In fact, the first two systems would be over-dimensioned. On the other hand, if we had assumed 'one omissive fault' and built the third system based on this assumption, it could fail if two omissive faults or one arbitrary fault occurred. The problematic we are addressing is generically called **coverage** of the fault tolerance mechanisms.

Coverage - given a fault in the system, coverage is the probability that it will be tolerated

It our context, it all boils down to **assumption coverage**. The assumption/coverage binomial is a quite important issue in the design of distributed fault-tolerant systems. When we assume that predicate $\mathcal{P}$ will hold with a coverage Pr, we say that we are confident that $\mathcal{P}$ has a probability Pr of holding. There is an important *separation of concerns* to be made in system design (Powell, 1992):

- **environmental assumptions** – the assumptions concerning the behavior of the environment where the system will run (which includes the infrastructure, networks, hardware, etc.), namely its faulty behavior;
- **operational assumptions** – the assumptions concerning the behavior of the system proper, or how the system will run (which includes programs, algorithms, protocols, etc.), under a given set of environmental assumptions.

The *environmental assumption coverage* (Pr_e) is then concerned with the conditional probability of a set of assumptions ($\mathcal{H}$) holding— such as clock rate of drift, network datagram delivery delay, omission error degree, number of component failures— given any occurrence of a fault f. Likewise, *operational assumption coverage* (Pr_o) is related with the probability that a given algorithm ($\mathcal{A}$) solves a problem, given the assumed set of environmental assumptions. Note that in fault-tolerant algorithms, this denotes the coverage of the error-processing mechanisms. Normally, if the algorithm and its implementation are proven correct, we expect a coverage $Pr_o = 1$. Pr_e is always an upper bound on total coverage. These are called deterministic algorithms, in contrast to probabilistic ones, where $Pr_o < 1$. In consequence, given any fault f:

$$\textit{if } Pr_e = Pr\{\mathcal{H}|f\} \text{ and } Pr_o = Pr\{\mathcal{A}|\mathcal{H}\} \text{ then } \textit{total coverage} \text{ is}$$
$$Pr_o \times Pr_e = Pr\{\mathcal{A}|f\}$$

For example, we make a set of assumptions $\mathcal{H}$ about the environment, say:

H1- during a reference interval T, at most k omissions occur

H2- the participants are not subjected to partitioning

Then, we design a deterministic reliable multicast protocol $\mathcal{A}$, which has several properties, for example:

A1- any message delivered to a correct participant is delivered to all correct participants

A2- any message sent by a correct participant is delivered to at least one participant

The architect should follow two complementary tracks in designing the system: (i) proving that the protocol, based on the set $\mathcal{H}$, ensures that every Ai holds with coverage one; (ii) determining the coverage of every Hi of $\mathcal{H}$; (iii) checking whether the total coverage, which equals the environmental assumption coverage Pr_e, is satisfactory with regard to the requirements.

6.1.5 How do computers fail?

An important and useful way of discovering which faults are relevant is to gather empiric data from systems already in operation. A study on the causes of system failures in large information systems conducted by Tandem Computers has been reported by (Gray, 1986). The results are quite interesting, and still relevant today. In this study, the smallest contributor to system failures were hardware faults. Most faults (42%) were caused by incorrect system administration or human operators (as we said, users do not always understand the system). Software faults were the second cause of failure (25%). The third contribution came from environmental causes: mainly power outages, but also a fire and a flood (disasters do happen!). Other studies confirm these numbers (Gray and Reuter, 1993; Pfister, 1998).

Several lessons can be extracted from these numbers, and these lessons can help the system architect in the task of building dependable systems. The first lesson is that system dependability can be increased by using appropriate administrative and system operating procedures. The second lesson is that software development methodologies that promote fault prevention and removal can also significantly increase system reliability. The third lesson is that software fault tolerance is a critical aspect in dependable computing. Although this book presents many useful abstractions for the development of distributed software, it is not specifically targeted at software engineering. However, the interested reader will find that many of the techniques described in these chapters can be used to tolerate both hardware and software faults.

6.2 FAULT-TOLERANT COMPUTING

As we have just seen, given the impossibility of avoiding all faults, one has to tolerate them in order to achieve dependable computing. Fault-tolerant computing refers to the techniques that can be used to prevent faults from generating failures. As you can imagine, there is no single technique that solves all problems: the most suitable set of techniques needs to be chosen depending on the classes of faults to be tolerated and the service requirements.

If the end user can easily tolerate a small down-time period, fault-tolerant techniques must prevent faults from creating an erroneous system response and support a quick repair of the failed components. Often, the easiest way of preventing a wrong result from being produced is to shutdown the system as soon as a safe state is reached. For instance, in a train control system a safe state can normally be reached simply by stopping all trains. In many other cases, continuity of service must be provided even in the presence of faults: an airplane cannot be stopped before landing; the downtime of a Web server may cause unacceptable loss of revenue. In all cases fault tolerance requires the use of redundant resources.

6.2.1 *Space, Time and Value Redundancy*

The use of *redundancy* is fundamental to fault tolerance. Redundancy assumes several facets, qualitative and quantitative. Since drivers do not want to be stopped on account of a flat tire most cars carry a spare one. Raiders will not want to be trapped in the middle of the Sahara Desert and so they will carry not just one but a few spare tires. Professional trucks will mount twin wheels, because the damaging of a tire while riding is itself unacceptable. In computer systems, redundancy can be applied in the space, time and value domains, and as much of it as needed.

Space redundancy consists of having several copies of the same component. The same information can be stored in several disks, tolerating the loss of one disk (if disks are placed very far apart, we can even tolerate events such as floods or fires). Different nodes can compute the same result in parallel to ensure that, even when one of them crashes, the result is available on time (active replication). In a distributed system, information can be disseminated along different network paths, to survive physical media damage.

Time redundancy consists in doing the same thing more than once, in the same or in different ways, until the desired effect is achieved. A simple example of time redundancy is the retransmission of a message in order to tolerate omissions due to electromagnetic noise or temporary receiver overflow. More sophisticated examples consist in repeating computations that have aborted because of temporary software faults (overload, particular interleaving of operations causing deadlock, etc).

Value redundancy consists in adding extra information about the value of the data being stored or sent. This extra information is normally control data in the form of codes that allow the detection, or even the correction, of integrity errors in the data being stored or transmitted. For instance, a parity bit or an error correcting code can be added to memory chips or to disk structures, respectively to detect or detect/correct data corruption. Frame check sequences or cyclic redundancy checks can be added to the data being transmitted in order to detect multi-bit corruption by noise. Cryptographic message signatures do the same in the presence of malicious errors.

6.2.2 *Error Processing*

Error processing has three facets: error detection, error recovery, and error masking. *Detection* is the first step at avoiding failure. Detection can be performed by several mechanisms, depending on the type of error: hardware bit-by-bit comparison; error detecting codes and signatures; timeouts or watchdogs; syntactic or semantic checks, and so forth. Once detected, an error can be confined, such that it does not propagate. If there is not enough redundancy in the system to recover from the error, the component or the system can at least be shut down, confining the behavior of the system to crash failures. For example, this characterizes the way self-checking components operate (Carter and Schneider, 1968; Wakerly, 1978).

A more effective approach is *error recovery*, which requires the system to have enough redundancy to carry on operating despite the error. There are two main approaches to error recovery. One consists in going back to a correct state and try to restart the computation from there. It is called *backward* error recovery. A crude version of this approach is emblematic of computer professionals: many jokes exist on the common belief that turning the computer off and on is *the* solution to most problems (unfortunately, things such as memory leaks tend to reinforce this belief). Fortunately, backward recovery actions need not be so drastic. For instance, detecting a frame corruption using the Cyclic-Redundancy-Check (CRC) and requesting a retransmission is a simple, common, and effective form of backward recovery. Another example consists in performing a computation and checking the result according to some pre-defined assertions; if the result is incorrect, it is ignored and an alternative algorithm is tried (this approach is known as recovery blocks (Randell, 1975)). However, as some of us have already discovered by personal experience, going back to a correct state is not always as trivial as it may look: (a) the error may have propagated to other components, making recovery hard if not impossible; (b) going back may require undoing (aborting) intermediate computations that may in turn have affected other computations; (c) the computation may have produced effects outside the system, that cannot be undone by the system alone. If these problems are not properly addressed, they may leave the system in an inconsistent state. One common form of backward error recovery involves progressing by steps in the computation, and storing the system state at the end of each step. The saved state is called a *checkpoint*. When an error occurs, the computation resumes from the latest checkpoint, after re-storing the relevant state.

The alternative to backward recovery is naturally *forward* recovery, which consists in taking corrective measures that cancel or alleviate the effects of the error. Forward error recovery is often a necessity. In some cases there is not enough time to go back to a correct state and to restart from there. For instance, if a message with a sensor reading is lost, it is preferable to wait for the next reading than to request the retransmission of the reading, which gets out of date. External actions that are impossible to undo also require some form of forward recovery when errors occur. For example, if the cash dispenser breaks just after the notes are handed to the customer but before the transaction is completed, it can no longer be rolled-back (that would be backward recovery). Instead, some form of mechanism must allow the customer and transaction records to be updated later. One form of recovering from crash failures in components is by reconfiguration, for example, *switching over* to a spare component. For example, in a dual-bus LAN, when one medium is detected failed, message exchange switches over to the other.

Inasmuch as cars have single wheels but trucks have twin wheels, having a pair of everything in a computer system is only cost-effective for special-purpose cases where interruption of service, even for brief moments, is not acceptable, and may even lead to damages far greater than the cost of the replica(s)— note

Table 6.3. Error Processing Techniques

error detection	detecting the error after it occurs aims at: confining it to avoid propagation; triggering error recovery mechanisms; triggering fault treatment mechanisms
error recovery	recovering from the error aims at: providing correct service despite the error
backward recovery:	the system goes back to a previous state known as correct and resumes
forward recovery:	the system proceeds forward to a state where correct provision of service can still be ensured
error masking	the system state has enough redundancy that the correct service can be provided without any noticeable glitch

that a pair of something only handles a single omissive fault; other types of faults require even more redundancy. Anyway, if enough redundancy is added to the system, errors can be automatically masked and never become visible, because there will always be a way of providing the correct result. This is *error masking* (also called *error compensation*). For instance, assume that you have a communication medium that exhibits omission faults, and you want to build a system that tolerates a single omission. If you can afford doubling the bandwidth and transmit every message twice, the error will be always masked. A similar approach can be used to handle crash failures of a single component: just use two components and pick the result that is available. If your car has twin wheels, one of the tires can tear and you can still continue to drive happily without even noticing the error (the "happy" epithet only applies to readers not living in the suburbs of a big city).

Summarizing these concepts in Table 6.3, we conclude with a few remarks:

- in some systems, error detection is just followed by isolation of the failed component, which implies that the system either gracefully degrades if the component is not vital, or is shut down if otherwise— self-checking components fall into this category;
- error recovery requires error detection, and the two make up the most used combination of error processing techniques;
- error masking does not require detection because it is applied systematically;
- however, error detection should always be used, such that the faulty component can endure *fault treatment*: isolation, removal, repair.

6.2.3 Evolution of Fault-Tolerant Computing

A non-exhaustive list of the major milestones in the evolution of fault tolerance (FT) in the past few years is given in Table 6.4. The list is necessarily incomplete, and tries to refer to the works more related to distributed systems. Fault-tolerant computing is historically associated with control applications. Its foundations lie in fault-tolerant electro-mechanic and digital design. Progressively, fault tolerance has occupied its place as a prominent design concept to improve dependability of computing systems in general, and of distributed systems in particular. *Hardware-based* fault tolerance was a pioneering concept, whose basic ideas were introduced by Von Neumann. It relied on *hardware component replication* techniques, which consist in using duplicate or triplicate components that operate in lock-step and whose results are filtered by (hardware) voting components. The idea behind such techniques is that the voting element, given its simplicity, can be made by design much more robust than the component being replicated. By construction, replicated components are usually tightly-coupled, often in the same physical board. Hardware-based fault tolerance can be an effective way of ensuring that some system component has a controlled failure mode (for instance, that it only fails by crashing or that it never exhibits Byzantine behavior). One of the first implementations of the concept in a computational system was the FTMP (Hopkins et al., 1978), used in a flight control system, where components were triplets working in lock-step, what was called triple-modular redundancy (TMR). However, the approach also has some obvious disadvantages:

- it does not provide the answer to faults that affect all replicas, such as catastrophes (e.g., floods or fire) or design errors;
- specialized hardware is not cost-effective and hard to update at the same pace of Commercial Off-The-Shelf (COTS) components;
- finally, hardware faults are just a small portion of the faults a system is subjected to, and this portion is becoming less and less significant given the complexity of today's software.

With the increment in the use of fault tolerance in computer systems in general, an obvious evolution consisted in resorting to *software component replication*. The use of such techniques forms what is called *software-based* fault tolerance, whose simplest form mimics the hardware FT approach. Instead of replicating hardware components, software components are replicated and their results consolidated by a voter component also built in software. Software-based fault tolerance can be more cost-effective than hardware-based FT. It is also simpler to add or remove software replicas than hardware replicas.

Software replicas can have varying granularity, from whole programs (e.g., database) to functions (e.g., cryptographic algorithm), and can provide varying levels of resilience within the same system, achieving what is called *incremental* fault tolerance. Besides, they can be executed in the same or separate hardware modules, which may co-reside (e.g., multiprocessors) or be distributed. This prefigures a new style of system design that may be called *modular* fault tol-

Table 6.4. · Major Milestones in Fault-Tolerant Computing

1967	Diagnosability in computer systems (Preparata et al., 1967)
1968	Self-Checking component (Carter and Schneider, 1968)
1975	Recovery blocks for software FT (Randell, 1975)
1976	Transactions (Eswaran et al., 1976)
1978	Distributed two-phase atomic commitment (Gray, 1978)
1978	TMR based computer system – FTMP (Hopkins et al., 1978)
1978	N-version programming (Chen and Avizienis, 1978)
1978	Byzantine agreement for FT communic. (Lamport et al., 1982)
1979	Weighted voting for replicated data management (Gifford, 1979)
1978	Distributed software-based FT – SIFT (Wensley et al., 1978)
1978	State machine FT programming model (Lamport, 1978a)
1981	Distributed Atomic Transactions (Lampson, 1981)
1985	Group-oriented FT programming model – ISIS (Birman and Joseph, 1987), AAS (Cristian et al., 1985)
1985	FT distributed-system fieldbus – MARS (Kopetz et al., 1989a)
1985	Commercial FT computers – Stratus (Wilson, 1985)
1986	Commercial FT computers – Tandem (Bartlett et al., 1987)
1986	Commercial LAN with medium FT – FDDI (FDDI, 1986)
1987	Distributed and incremental FT (Powell et al., 1995)

erance: system architects break down the system in software modules, decide the different levels of replication of each module, and foresee the number and placement of the necessary hardware modules where software modules will be installed.

The most obvious use for this modularity is *distributed* fault tolerance, which leverages the advantages of modular fault tolerance through the failure independence given by geographical dispersion: each replica of a given component is located in a different node of a distributed system. Modular and/or distributed FT are the most used approaches today in dependable system design. The emphasis on component interaction explains why algorithms and protocols are so relevant in this class of systems, and also why the focus lies on interaction faults, as discussed earlier. In addition to these important arguments, there is today a fundamental case in using distributed fault tolerance: existing computing systems are distributed and need to be made dependable. Distributed FT provides the ground to tolerate several classes of faults:

- hardware faults, since the results of the several replicas are consolidated such that a correct value is returned, despite the failure of hardware components;
- transient software faults (also called Heisenbugs from the Heisenberg uncertainty principle), since they occur sporadically and are thus normally masked by redundant execution in different environments;
- disasters, since the geographic dispersion of nodes supported by distributed FT provides the basis to survive them.

In the case of design faults, simple replication provides little help: errors will systematically occur in all replicas. In consequence, what is needed is that replicas are designed diversely, different system architectures are used, or execution results are tested against assertions about the desired outcome. For example, each replica of a given component can be designed and developed by a different team of programmers. This is one of the main techniques to achieve *software fault tolerance*. Software design diversity is rather expensive (most software products already cost too much, even when a single development team is involved) and as such it is only employed when the stakes are very high, such as in safety-critical systems. On the other hand, a characteristic of distributed fault tolerance is that replicas of a same component can reside in different hardware and/or operating system architectures, and execute at different moments and in different contexts. This implicit "diversity" is enough to tolerate many design faults, especially those hardware or software faults that lead to an intermittent behavior. It follows that (external) transient faults can also be tolerated this way.

6.3 DISTRIBUTED FAULT TOLERANCE

When characterizing distributed systems in general (and not necessarily fault-tolerant) we have mentioned a number of important properties, such as modularity, support for heterogeneous hardware, incremental growth, etc. These properties derive from the distributed nature of the system and are not specific to fault tolerance. However, they are also extremely important when applied to fault tolerance! We will illustrate this fact with a couple of examples.

An important property for fault tolerance is *modularity*. Distributed fault tolerant systems are built of nodes, networks and software components. Failure assumptions are mainly concerned with the interactions between these hardware and software components. Construction of these systems is based on modular hardware and software units. A separation of concerns is sought by attempting at decoupling software units from the hardware units where they execute, as suggested by Figure 6.3. Fault tolerance is to a large extent achieved through adequate protocols to govern the interactions between components in the presence of the assumed failure modes.

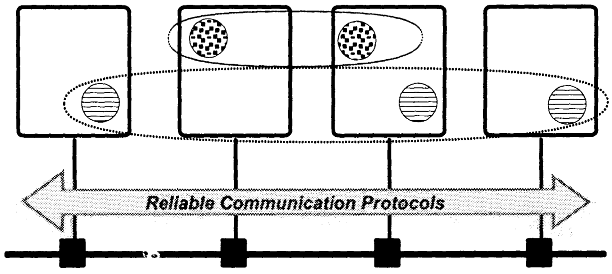


Figure 6.3. The Principle of Distributed Fault Tolerance

If the appropriate design techniques are used, either hardware or software components can be replaced without changing the complete architecture. In result of this modularity, different dependability levels can be achieved using different combinations of components and protocols. A distributed architecture can thus provide *incremental dependability*, i.e., higher dependability features can be obtained in an incremental way: by enhancing the more fragile components (fault prevention); and/or incrementing the number of replicas of each component or making them resilient to more severe faults (fault tolerance). Another advantage of modularity lies in making it easier to build systems that exhibit *graceful degradation*: when some components fail, instead of collapsing, the system continues providing a lower level of service. It is possible to progressively decrease the degree of dependability of an application, or to preserve some applications in detriment of others.

Support for *heterogeneity* is also a key element of distributed fault tolerance. As we have mentioned before, it allows the use of components with diverse design and this can help in tolerating design faults. Support for different hardware also simplifies the evolution of the system, puts less constraints when buying new hardware, and allows to minimize the financial costs associated with the use of redundancy. On the other hand, it enhances *maintainability*, through the possibility of re-instantiating failed software units in available hardware units of different makes.

It is also easy to extend some of the fundamental properties of distributed systems to make them useful for fault tolerance. We will just give one example using the *encapsulation* property. Distributed systems allow designers to assume a glass-box view of interconnected black-box components. Remote object invocation is a simple way of hiding the internal structure of the server from clients. This property allows the system architect to build modular, possibly distributed, fault-tolerant subsystems, which she may recursively use as black-box components with resilient properties at a higher level of abstraction. For example, a closely-coupled multicomputer may be built with distributed fault tolerance techniques, and then used as a resilient black-box component of a wider fault-tolerant distributed system.

We must end this section with a word of caution. Despite all these advantages, the system architect should never forget that complexity is itself a potential cause of faults. We have started this chapter by stating that many faults are due to unexpected or not fully understood interactions between different components of the computing system. Modular and distributed systems do not make these interactions any simpler. The KISS rule should be considered as a rule of thumb by every architect: "Keep It Simple, Stupid!" On the other hand, rigorous design principles must be followed when building a dependable distributed system. Chapters 7 and 8 will survey the main paradigms and models than can help the system architect in this task.

6.4 FAULT-TOLERANT NETWORKS

We have just mentioned several advantages of distributed fault tolerance. Before you go over-enthusiastic about this strategy, remember that there is usually no such thing as a free lunch. The "catch" here is that distributed components need a communication network to interact with each other. Naturally, the communication network should not be a single point of failure itself. Thus, we need to build *fault-tolerant networks* too!

The interaction failures that we studied earlier are an adequate representation of what may go wrong in networks: omissions due to lost messages (messages can be lost because of noise, lack of clock synchronization, overflow at the recipient or in a router, etc); timing failures, specially when a single channel is shared by several nodes; assertive failures, when data is corrupted along the transmission path.

A belt-and-suspenders approach consists in constructing a fully space-redundant architecture. Each node is connected to the other nodes by more than one link in parallel. The example illustrated in Figure 6.4a features a duplicated broadcast bus LAN, where we can see that the complete network is replicated, including the physical channel and the communication boards. This architecture tolerates one omissive fault. The actual number of network replicas will depend on the number and type of assumed faults. Variants may exist depending on the criticality. For example, the buses may be unidirectional (single transmitter, multiple receivers), one per node, so that no one node can disrupt communication by jabbering the channel. This approach would require four channels for the example in Figure 6.4a. Alternatively, the network may have the topology of a completely or partially-connected graph of point-to-point links, where reliability is achieved through store-and-forward transmission through the several alternative routes, as exemplified Figure 6.4b. Hypercubes are a common topology in this kind of networks. The fully space-redundant architecture provides a basis to achieve tolerance of any class of fault. However, it is very expensive and is only used in extreme cases.

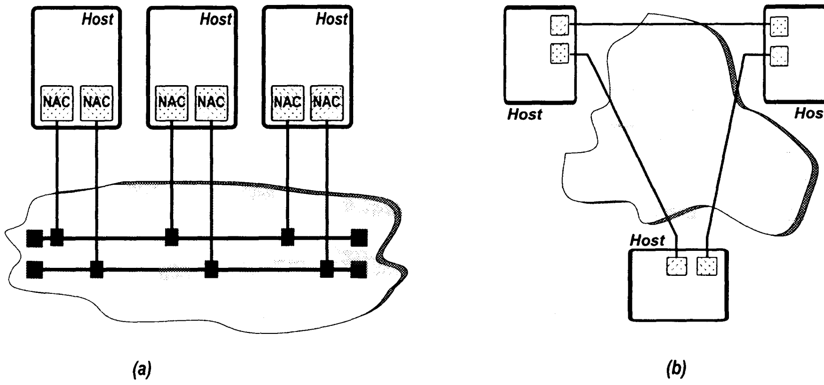


Figure 6.4. Space-Redundant Network Architecture: (a) Bus; (b) Point-to-Point

For omissive faults, there is also the alternative of using time redundancy, i.e., to send the same message several times through a simplex (non-replicated) channel. However, even in this simple case, if components fail permanently (e.g., crash of an Ethernet hub), or exhibit a large number of errors (e.g., frame omission errors due to environmental noise), time redundancy is not enough. An intermediate approach is depicted in Figure 6.5. The medium-redundant architecture aims at achieving a non-stop medium and provides an extremely cost-effective solution to the fault-tolerant communication problem. Replication is done only for the physical layer components of the architecture (cabling, modems, codecs, transceivers). The upper layers do not even known that such redundancy exists, and so any protocols for simplex networks can be used transparently. Figure 6.5a presents a dual- medium bus architecture, where transmission is done on both media, but reception is switched over to the alternative medium, upon detection of a medium failure, on a per recipient basis, a shown. On the other hand, Figure 6.5b presents a dual ring architecture. Communication takes place in the primary ring (outer one), which reconfigures if any of its parts fails and interrupts the ring, by wrapping around the secondary ring, as shown in the figure.

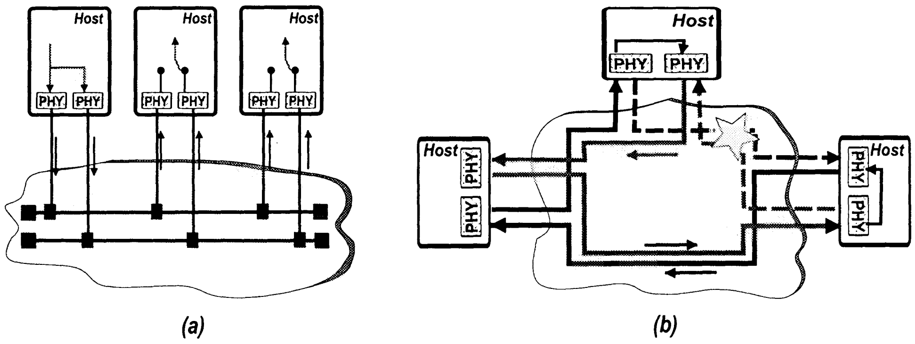


Figure 6.5. Medium-Redundant Network Architecture: (a) Bus; (b) Ring

6.5 FAULT-TOLERANT ARCHITECTURES

We have been discussing several concepts related to building fault-tolerant systems. Let us illustrate how these concepts can be put into practice, by doing an overview of the main fault-tolerant computing architectures. The detailed paradigms and mechanisms that make these architectures work will be discussed in the subsequent chapters.

Figure 6.6 exemplifies two basic architectural approaches for achieving local availability. Figure 6.6a exemplifies *redundant storage*, which achieves availability through disk replication, for example by means of RAIDs (Redundant Arrays of Inexpensive Disks), redundant disks that provide several levels of reliability. The highest level can guarantee virtually non-stop operation. However, such architectures do not solve the problem of processor failures. Figure 6.6b depicts the *redundant processor* approach, whereby the computer can be provided with more than one processor module or board, so that it remains available in the case of one or more processor failures. A dual processor architecture provides a comfortable level of availability, and is often seen combined with the RAID-based redundant storage approach to deploy highly-available servers.

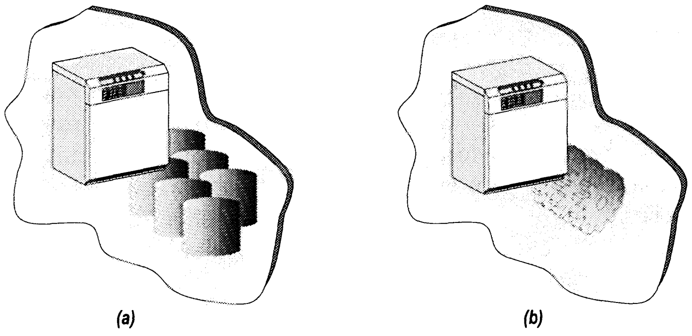


Figure 6.6. Redundant Architectural Modules: (a) Storage; (b) Processors

These basic architectures remedy the problem of crashing components, but say nothing about their possible misbehavior. Figure 6.7a illustrates the *self-checking* architectural approach for achieving local reliability vs. a wide set of faults, even assertive ones. The idea underlying the approach is that there is a component (the checker) which performs surveillance of the unit (which by the way can be a whole processor). The checker analyzes all operations and immediately stops the unit when it detects an error, before an erroneous result is propagated. A unit that is not allowed to do any errors whatsoever while functioning is said to be *fail-silent*: it behaves correctly or else fails by crashing. Now we have guaranteed correctness, but lost availability. Figure 6.7b depicts a well-known approach to achieve reliable and available processing. *N-modular redundancy*, or *NMR*, is a concept whereby several modules execute identical steps, in a tightly synchronized (lock-step) manner, so that results can be compared on a bit-by-bit basis by a voter. As long as there are less than half failed modules, the unit executes correctly, whether components crash or produce incorrect results. The example in the figure is a triple-modular redundant (TMR) architecture, which tolerates one faulty module. Note that a self-checking unit can also be built out of a 2-MR unit.

These basic concepts can be combined and applied in a broader and distributed context. For example, the lock-step model is too constraining. However, the same concept can be applied to *distributed replicated processing*, as depicted in Figure 6.8. The foundations of replication as a distributed activity have been discussed earlier (*see Replication* in Chapter 3). Based on distributed fault tolerance, these architectures provide great versatility. As exemplified in the figure, a number of replicas residing in different sites perform replicated computations, and interact among themselves through protocols that ensure resilience to the assumed fault classes (e.g., timing, value, arbitrary, etc.). The number of replicas required varies according to the fault model: class and number of faults assumed. For example, in order to tolerate one crash fault, two replicas are needed and the first result is taken. Tolerating one value fault

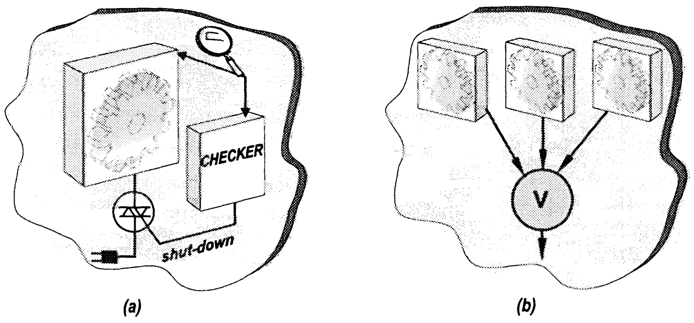


Figure 6.7. Redundant Architectural Modules: (a) Self Checking; (b) NMR

requires a majority vote between three replicas. If faults are arbitrary, then at least four replicas are required. Replicas receive inputs, execute processing steps and produce outputs in the form of messages. This is also called a *state machine* model. Unlike lock-step execution, replicas can execute at slightly different times, and in different ways, if the hosts are heterogeneous. However, they should produce the same outputs, for a same sequence of inputs.

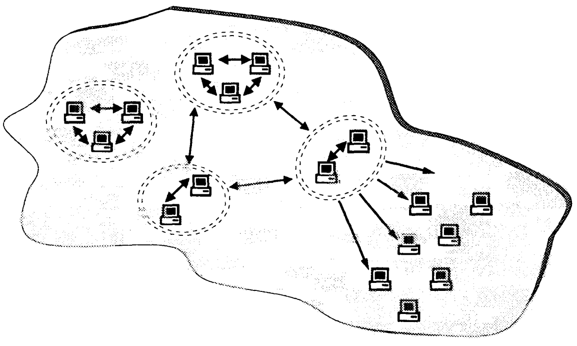


Figure 6.8. Distributed Replicated Processing Architecture

As suggested in Figure 6.8, different replica groups may communicate among themselves in order to construct more elaborate distributed computing architectures. Some architectures studied in Part I of this book (*see Distributed System Architectures* in Chapter 1) can benefit from such combinations of distributed fault tolerance techniques in order to become dependable. Figure 6.9 gives two important examples: client-server and publisher-subscriber. Figure 6.9a exemplifies how to render a service dependable. The underlying *logical* server is in fact made of two or more replicas. The principle of operation is that client requests are addressed to all replicas, executed by all, and a consolidated reply is given back to the client. The client need not (should not) even known that the servers are replicated. This is called *replication transparency.* The important notion is that the service remains available despite failing servers.

Figure 6.9b illustrates a dependable publisher-subscriber system. Recall that the architecture relies on a network publishing server. This server is a single point of failure, which is rather awkward given the notion of "bus" purported by the underlying model. That can be avoided if the server is replicated. The more widely it is replicated, the more it resembles a bus, in the sense of making information reach everywhere and not depending on the failure of a single component. Information is published to all replicas, for example through multicast. In absence of failures, subscribers may get their information from different servers that share the publishing load.

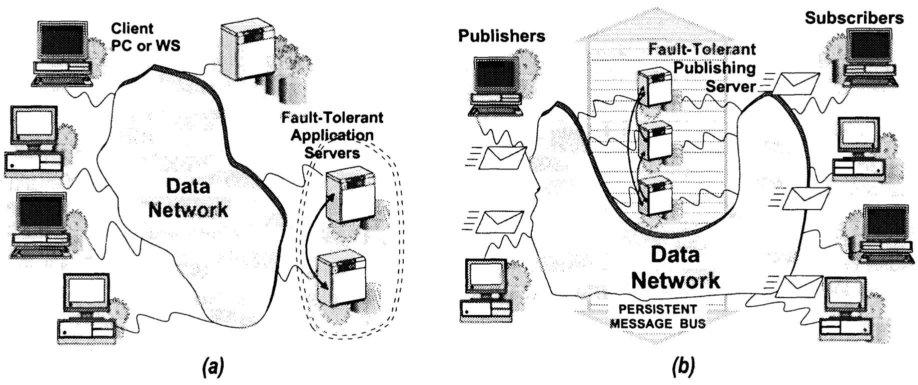


Figure 6.9. Distributed Fault-Tolerant Architectures: (a) Client-Server; (b) Publisher-Subscriber

6.6 SUMMARY AND FURTHER READING

Fault tolerance is fundamental to building dependable systems, i.e., systems that we can depend upon. This chapter has reviewed the basic concepts and terminology underlying dependability, and introduced the fundamental approaches to fault tolerance. Finally, we have characterized distributed fault tolerance and addressed the motivations for using the approach, which encompass both making distributed systems dependable and making dependable systems using distribution.

The seminal work on dependability concepts of Laprie, in the context of the IFIP WG10.4 on Fault Tolerance, is a must read for everyone working in fault tolerance (Laprie, 1987; Laprie, 1992). Other relevant works include the good survey of early fault-tolerant systems provided by Rennels (Rennels, 1984), or the more recent surveys of (Cristian, 1994) and (Mishra and Schlichting, 1992; Powell, 1994). A discussion on failure mode assumptions and assumption coverage is given in (Powell, 1992). For some works giving a comprehensive treatment of other aspects of dependability besides distributed fault tolerance, see (Lee and Anderson, 1990; Jalote, 1994; Laprie, 1998).

7 PARADIGMS FOR DISTRIBUTED FAULT TOLERANCE

This chapter discusses the main paradigms concerning fault tolerance in distributed systems. Namely, the chapter addresses: failure detection, membership, fault-tolerant communication, replication management, resilience and recovery. The paradigms are explained in practical terms, by exemplifying the problems they solve, as well as their limitations.

7.1 FAILURE DETECTION

We have seen previously that one approach to build dependable systems is to detect an error and later recover from it. Since an error arises from a failure occurring at component level, component *failure detection* is fundamental to fault tolerance. Even if the system is able to mask the error, failure detection pins down the affected component. The failed component can then be disconnected or repaired, and the desired level of redundancy restored in the system. Failure detection is also important from a performance viewpoint: if a component is known to be failed there is no point in wasting resources trying to communicate with it. This section discusses the aspects related to failure detection in modular and distributed systems. As it will be seen, distributed failure detection is harder than it might look at first glance. Additionally, accurate failure detection is impossible in systems where the network can partition.

In order to detect the failure of a given component (the target) we need another component (the failure detector). We also need some channel between the failure detector and the target component such that the behavior of the latter can be monitored. Thus, in order to detect the failure of a component we need to add two more components to the system, and these components may also fail!

One of the possible outputs of a faulty failure detector consists in marking a correct target component as failed. That is why failure detection is a complex issue. Usually, the failure detector should be constructed in a way such that it exhibits a much higher reliability than the observed component (it should be much simpler, or be made of better hardware, or both). Additionally, the system should be constructed in such a way that the consequences of erroneous failure detection are less severe than the absence of failure detection. For instance, consider that a signaling system is introduced in a railway system to prevent train collisions. If because of benign failures in the signaling system the trains are forced to a brief halt once in a while, this is still a reasonable price to pay to avoid the loss of human lives (after all, better to arrive later than never).

Depending on its properties, the channel between the failure detector and the target can also complicate the task of achieving reliable failure detection. If the channel is not perfect (i.e., it may lose or delay information), then it may be difficult, if not impossible, to distinguish the failure of the observed component from the imperfect behavior of the channel.

7.1.1 Local Failure Detection

Let us start with local failure detection. By "local" we mean failure detection in an environment where the detector and the target are "close" enough to establish a "perfect" observing channel. For instance, the failure detector component may be in the same machine or even in the same board of the target component so that reliable communication mechanisms exist (O.S. or HW) to establish the observation channel.

Examples of local failure detection are *self-checking* routines, implemented in software or hardware such as parity checks (in memory, disks or buses). *Guardian* components check the validity of the outputs produced by the observed component: for instance, they can check if memory accesses are performed within some pre-defined allowed range. This type of failure detection is reliable, given that the channel can be considered perfect and the failure detector is quite simple. Other examples of local failure detectors are *watchdog* components. They test whether a computation progresses within a certain pace. They can be implemented in software or hardware. A HW watchdog is a down counter fed by a clock. It is loaded with the equivalent of a time interval, and the observed process has to reset it before it expires, otherwise a failure signal is produced. A SW watchdog may be implemented by the O.S. For instance, a process is instrumented to periodically set a memory position at certain points in the computation, to show it is making progress. The O.S. periodically verifies the position and resets it. Whenever it finds the position not set, it produces a failure indication (the process is late or lost). If the O.S.

is capable of controlling its timeliness, these failure detectors can be considered reliable.

However, even in a local environment failure detection may be harder than it looks. Consider the software watchdog we have just mentioned. If the monitoring is not performed by the kernel itself but by another user-level task instead, that task may not have enough information about the system scheduling decisions to distinguish the failure of a process from a load problem: the machine may be overloaded and the target process may have been swapped-out and not scheduled for some time (therefore, unable to update the variable) even though it is still in a correct state.

7.1.2 System Diagnosis

The previous model considered two different types of system components: the target components and the failure detectors. One can generalize this model by considering all components alike: each component plays a dual role in the system, providing service and testing other components. *System diagnosis* consists in identifying which system components are faulty based on the results of tests that components mutually perform on each other. Let us start with the assumption that the outcome of tests reported by correct components can be trusted. On the other hand, faulty components may incorrectly report correct components as faulty (or faulty components as correct). The difficulty here is that there is no *a priori* knowledge of which components are faulty, so the diagnosis has to be performed by analyzing the reports from all components.

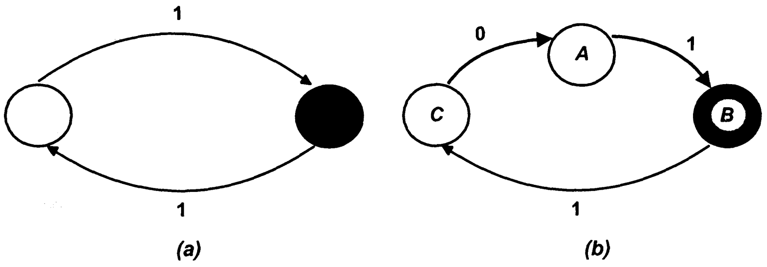


Figure 7.1. System Diagnosis: (a) Symmetric Detection (one faulty); (b) Diagnosis Ring

One way to represent a system for diagnosis purposes is to use a directed graph, where nodes represent components and edges link observers to observed nodes, in this direction. The label on the edge is either correct (0) or faulty (1). Faulty nodes are represented in black and correct nodes in white. Consider the simple example of Figure 7.1a with just two nodes that observe each other. In the example both nodes have marked the peer as faulty! Since the arcs in the Figure are symmetric, without *a priori* knowledge it is impossible to assess which is the faulty component. In fact, it has been shown (Preparata et al., 1967) that in a system where f components may be faulty we need: $n \geq 2f + 1$

processes to diagnose the fault; and that each component is tested by at least f other components.

Consider now the example of Figure 7.1b, where alternatively the diagnosis graph is organized in a ring. If there is at most one faulty process, it can always be identified, since there are three nodes. The faulty process will have: (i) a converging edge marked 'faulty'; and (ii) the source node of that edge marked 'correct'. B obeys these conditions, C is still considered correct despite marked faulty by B (because the latter does not obey (ii)). In order to perform the diagnosis, the failure detection reports must be collected and analyzed by a (logically or physically) centralized component, usually a supervisor external to the system.

7.1.3 Distributed Failure Detection

Distributed failure detection is harder than local failure detection, basically because it has to rely on message exchange (no shared memory or local interconnection buses are available). Thus, the communication channel between the observer and the target may not be perfect.

In order to focus on the difficulties introduced by distribution we will now assume that we are attempting to detect the failure of processes. To make things even simpler, we assume that processes can only fail by crashing and that the system is synchronous (i.e., delays are bounded). Thus, a process is correct as long as it provides evidence of activity (by sending or replying to messages). How this activity is monitored depends on the detection protocol. It may expect that the observed component periodically sends messages on the channel, usually called "I'm alive messages" or *heartbeats*. When these messages are missing the component is considered failed. Heartbeats are sent spontaneously, but an alternative form is for the monitored component to wait for a *probe* message coming from the failure detector and then reply with the "I'm alive" message. During periods of activity, message exchanges on behalf of the computation can be used to perform failure detection; special messages only need to be sent when the component is in an idle state. Practical systems can further optimize this process. For instance, if a broadcast channel is available, heartbeats can be sent in broadcast mode. Alternatively, the processes can be organized in a logical ring, and exchange heartbeats with their neighbors (this requires some re-organization in case of failure but minimizes the network traffic).

To simplify this even further, let us also assume that the network provides full connectivity, i.e., any process can send (and receive) messages directly to (and from) any other process. This is not always the case but allows us to abstract from the way nodes are connected at the network level. Using this topology, any process plays the role of an observer (to monitor the activity of other processes) and a target (i.e., it is monitored by all the other processes). Thus, instead of a one-to-one relation we have a many-to-many relation. Ideally, failure detection should be consistent; for instance, if a process fails, it should be detected as failed by all correct processes in the system. In a seminal work, Chandra and Toueg (Chandra and Toueg, 1996) have defined two properties that help formalizing this intuitive notion of consistency of distributed failure detection:

Strong Accuracy - a safety requirement, specifying that no correct process is ever considered failed

Strong Completeness - a liveness requirement, specifying that a failure must be eventually detected by every correct process

If perfect channels are available, heartbeat exchanges meet strong accuracy and strong completeness. Such a failure detector is called *perfect*. If a node crashes all correct nodes will note the absence of the heartbeat and will detect the failure.

What happens when the channel that interconnects the processes is not perfect? One must distinguish the case where the imperfection can be fixed by some simple protocol, from the case where the imperfection is impossible to overcome.

The first case is simpler to discuss. Assume that the communication channel is not perfect but has some mild imperfection, for instance, it makes a small number k of omissions. The solution to this problem is to transform the imperfect channel into a perfect channel using one of the redundancy approaches described in the previous chapter. For instance, each heartbeat can be retransmitted $k + 1$ times, effectively ensuring that it is observed by all correct processes.

7.1.4 When Failure Detection is Imperfect

Perfect failure detectors are obviously very convenient. When a process goes down all the other processes know about it and can coordinate their actions to implement corrective measures. Unfortunately, it is not always possible to implement perfect failure detection, and this is the harder case.

There are two major adversaries of perfect failure detection. One is the lack of bounds on the number and type of faults the communication channel may give. Imagine that the number of omission faults of a channel between two processes cannot be bounded. If a process does not receive any heartbeat message from the other process this may be because the other process is failed or because the channel has dropped all heartbeats sent so far. Actually, no type of coordination can be achieved between two processes if the behavior of the cannel is not restricted in some way. At least, the channel should not always drop all messages (or all messages of a certain type). Thus, it is usually assumed that the channels are *fair*, i.e., if a message is sent infinitely often by a process then it is received infinitely often by its receiver (Lynch, 1996).

A particular case of link failure that prevents any sort of failure detection occurs when the link crashes and one or several processes become disconnected from the rest of the network. In this case we say that *network partitioning* has occurred and it makes more sense to use the words *reachability detection* than failure detection. Note that consistency of reachability information is as relevant as the consistency of failure detection.

The other adversary of perfect failure detection is the lack of bounds for the timely behavior of system components (processes or links). If a link can delay a message arbitrarily, or if a process can take an arbitrary amount of time to make a processing step, there is no way to distinguish a missing heartbeat from an "extremely slow" heartbeat. This means that if the system is asynchronous, perfect failure detection cannot be implemented (*see Asynchronous Models* in Chapter 3).

This is a not a comforting conclusion. Distributed systems are complex enough even with perfect failure detection. Without it, most problems become much harder. Let us consider for instance the problem of having process A send a message m to process B over a lossy link. Assume that you define reliable communication in the following way: as long as A and B remain correct, A will succeed in sending m to B. A simple positive acknowledgement protocol can be implemented: A will re-transmit m until it gets an acknowledgement from B or until B crashes. However, without perfect failure detection, A will never be sure that B has in fact failed and, in order to meet the specification, it will have to store and retransmit m forever!

The impossibility of perfect failure detection raises another question: is there a middle term between perfect failure detection and no failure detection at all? Chandra and Toueg (Chandra and Toueg, 1996) have defined weaker forms of the accuracy and completeness properties:

Weak Accuracy - at least one correct process is never considered failed by all correct processes

Weak Completeness - a failure must be eventually detected by at least one correct process

Different classes of failure detectors can be defined combining weak and strong accuracy and completeness properties. Are all these classes useful? Let us discuss this issue in the context of pure asynchronous systems.

7.1.5 Asynchronous Failure Detection

We have just seen that the asynchrony of the system makes perfect failure detection impossible. Actually, it has been shown that several other problems requiring some form of coordination, such as consensus, atomic broadcast or atomic commitment have no deterministic solution in asynchronous systems subjected to failures. This is known in academia as the *FLP result*, after a famous paper, by Fischer, Lynch and Paterson that demonstrated the impossibility result for the consensus problem (Fischer et al., 1985).

Nevertheless, given that it is often impossible to impose a bound on message or processing delays, solutions for distributed coordination problems in asynchronous systems have been sought by several researchers. This effort lead to the following interesting question: what are exactly the minimum synchrony requirements to solve problems such as consensus? Chandra and Toueg showed that consensus can be solved in asynchronous systems augmented with failure detectors and that the weakest failure detector to solve the consensus problem has the following two properties (the solution also requires that a majority of processes are correct, and that no partitioning occurs):

Eventual Weak Accuracy - there is a time after which some correct process is never suspected by any correct process

Weak Completeness - a failure must be eventually detected by at least one correct process

A failure detector with these properties is called *eventually weak*. The reader should note that this definition only requires weak accuracy to be satisfied at some point in time. The intuition behind this requirement is that consensus can be solved if a period of stability is preserved long enough to allow coordination among the processes. Stability is defined in terms of having at least one correct process that is not suspected by any of the other correct processes. The intuition behind this is that during the stability period, this process can act as a coordinator that supports the establishment of consensus.

Naturally, the impossibility results still holds. This means that even a eventually weak failure detector cannot be implemented in a pure asynchronous system. This fact rose several doubts about the practical utility of the model. On the other hand, most of the algorithms having this model in mind, make so little assumptions about the system behavior that they never risk violating safety conditions even when the failure detector does not satisfy the above properties. For instance, in the algorithm proposed by Chandra and Toueg (Chandra and Toueg, 1996), if the failure detector does not satisfy its properties consensus may never be reached, but on the other hand the processes never take inconsistent decisions.

7.1.6 The Problem of Partitioning

Partitioning is caused by the crash of one or more links that split the network in disjoint subsets, or *partitions*. Processes within the same partition are able to communicate among themselves but unable to communicate with processes in other partitions. Network partitioning is a serious problem in distributed systems because it prevents processes in different partitions from coordinating their activities.

There are two main approaches to address the problem of partitioning. One is to allow uncoordinated progress in different partitions. When partitioning is *healed*, in other words, when partitions merge, processes have to *reconcile* their state. Automatic reconciliation is usually very difficult (or even impossible) in the general case. An alternative approach is to allow progress in one partition exclusively, the so-called *primary* partition. The primary partition approach prevents divergence (*see Primary Partition* in Chapter 2), but in turn it blocks all the other system partitions. Several different criteria can be applied to select the primary partition, one of the simplest being a majority criterion, as illustrated in Figure 7.2. It should be noted that the network can be partitioned in such a way that no primary partition can be identified and the system be forced to block until the partitions merge.

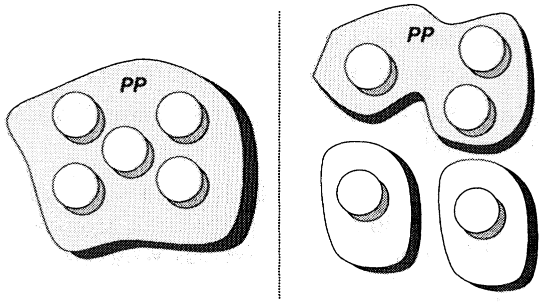


Figure 7.2. Partitioned Network with Primary Partition (PP)

Link failures are usually detected by timeouts (note however that some particular types of networks can provide accurate information about the state of links). This means that partitions can only be detected accurately if there are bounds on processing/communication delays and on the number of omissions tolerated. Otherwise, it may be impossible to distinguish a failed link from an extremely noisy or extremely slow link. It follows that partitions cannot be accurately detected on pure asynchronous systems. Inaccurate detections may create what is called a *virtual partition.* Participating processes behave as if the system was split, while in fact the links are just slow.

Conceptually, there is little difference between a virtual and a physical partition, since from the point of view of the participating processes, the two types of events cannot usually be distinguished. In practical systems, there is usually a huge difference. Virtual partitions tend to have a short duty cycle, regenerating rapidly and spontaneously, and may be repetitive. For instance a link can be temporarily overloaded because of an ongoing bulk-data transfer and regain its normal latency as soon as the data transfer finishes. However, a router may be on the verge of thrashing, holding long queues, discarding packets, recovering, degrading again, and so forth. The problem is that this may generate periods of *instability* where links switch from down to up state, forcing the system to be in permanent reconfiguration.

The application itself can cause this unstable behavior. Consider that for some reason, a link is so slow that it seems to be down. This creates a virtual partition in a distributed computation. When the link recovers, the partition is healed. Processes from either side start a state-transfer procedure to become mutually consistent. The state being transferred overloads the link and the virtual partition occurs again, aborting the state-transfer. This decreases the load on the link, causing the link to recover once more. The state transfer restarts, re-initiating the cycle. A behavior very close to this has been observed in early implementations of the BGP Internet routing protocol (Stewart, 1999).

The problem has no solution, but practical systems can alleviate these symptoms, in essentially two ways. Firstly, by making a best effort to tell physical partitions from virtual ones, so as to delay failure detection decisions to when there is a high certainty of actual failure. Quality of service failure detection is a possible approach along this line (*see* Section 13.11 in Chapter 13). Secondly, by devising programming models that accommodate partitioning. This will be discussed in subsequent sections of this chapter.

7.2 FAULT-TOLERANT CONSENSUS

We have introduced the *consensus* problem in the Distributed Systems Part of this book (*see Distributed Consensus* in Chapter 2). We recall here the definition of the consensus problem: each process proposes an initial value to the others, and, despite failures, all correct processes have to agree on a common value (called decision value), which has to be one of the proposed values. This problem is of paramount importance in distributed systems, particularly in fault-tolerant distributed systems, since many problems can be solved using consensus as a building block, for instance: membership (agreement on who are the members of a group), ordering of messages (agreement on sequence numbers), atomic commitment (agreement on the outcome of a transaction, etc). It is worth noting that the system diagnosis problem that we have addressed previously can also be considered a consensus problem, where correct processes must agree on which processes are faulty (for a deeper survey relating system diagnosis with the consensus problem see (Barborak et al., 1993)).

The solution to the consensus problem in a system where no faults occur is deceptively simple. For instance, consider the following solution: the processes with the lowest identifier is statically selected as the coordinator and sends its initial value to all the remaining processes; this value is the outcome of the consensus! In the absence of faults, this trivial protocol clearly solves the problem, since every process decides on a value and that value is one of the initial values. Curiously, it is extremely difficult to "extend" this solution to tolerate faults, even if a perfect failure detector is available. Let us see why.

To start with, it should be obvious that this solution is inherently non fault-tolerant since it relies on a single and fixed coordinator. If the coordinator crashes, the algorithm blocks. Note also that if the coordinator crashes while disseminating its value, some process may decide while others may remain blocked. It is tempting to believe that the problem can be simply fixed by selecting another coordinator in case the first one crashes. The updated protocol could work like this. When the failure of the current coordinator is signalled to the next process down the line (there is a total order on process identifiers), this process assumes the role of the coordinator and sends its initial value to every other process. When the same notification is received by some process other than the next coordinator, it simply waits for the value from the new coordinator. Unfortunately, this algorithm only works if the first coordinator does not crash during the dissemination of its value. Otherwise, some process may receive the value from the first coordinator (and decide on that value) and others from the next coordinator (and decide differently), which violates consensus. The trick is that a protocol to solve consensus has to prevent a process from deciding a value while there is no guarantee that this is the only value that can be decided by other processes, even if faults occur. When one is sure that a given value is going to be decided (even if not every process has decided yet), the value is said to be *locked.*

How can a value be locked? Consider that we add the following rules: when a process receives the initial value from the coordinator, it changes its initial value to that of the coordinator. Thus, if that same process later assumes the role of coordinator it proposes the value it has received from the previous coordinator (note that this does not prevent a process from proposing its initial value in the case it did not receive any value from previous coordinator(s)). The protocol can then be improved as follows. The coordinator sends its value to every other process. These processes do not immediately decide the value; instead, they update their initial value and send an acknowledgment back to the coordinator. When the coordinator receives an acknowledgment from every non-crashed process, it knows the value is locked. Even if it crashes, the new coordinator (one of the processes that have acknowledged) will also propose that same value, as illustrated in Figure 7.3. Of course, at this point only the coordinator knows that the value has been decided, so it disseminates a special DECIDED message to inform the remaining processes of that fact. When DECIDED is received from the coordinator, a process can safely decide on that value.

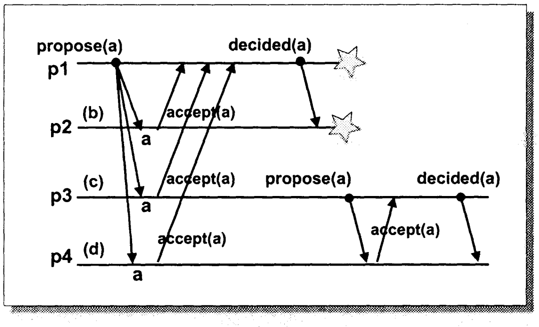


Figure 7.3. Fault-Tolerant Consensus (Perfect Failure Detector)

Note that the previous solution only works if failure detection is reliable. As we have discussed in the previous section on failure detection, in systems where failure detection is unreliable, namely in fully asynchronous systems, there is no deterministic solution to the problem (Fischer et al., 1985). We have also noted that consensus can be solved in an asynchronous system augmented with an eventually weak failure detector, as long as a majority of processes do not crash. We give a brief intuition of a possible solution. The protocol is quite similar to the protocol we have described above. However, the coordinator simply waits for a majority of acknowledgments to lock a value. This allows the system to make progress as long as a majority of processes can communicate, no matter whether the remaining processes are crashed or simply slow. On the other hand, when another process decides to become the coordinator its task becomes more complicated, since the previous coordinator may have locked a value without the intervention of the new coordinator. To avoid proposing an inconsistent value, the new coordinator has to contact a majority of processes in order to "check" whether a previously locked value exists.

7.3 UNIFORMITY

The fault-tolerant consensus problem can be defined with two distinct flavors: the *uniform* consensus and the *non-uniform* consensus. The uniform consensus definition states that if two processes decide, they decide the same value. Note that no distinction is made between faulty and correct process (the property applies to all processes in a uniform manner). For instance, if a process decides on a value and later crashes, all the correct processes must also decide on that same value. The non-uniform flavor is a weaker form of consensus stating that if two *correct* processes decide, they decide the same value. This allows processes that remain correct to decide on a different value than that decided by a crashed process.

Note the example of non-uniformity in Figure 7.4: suppose p and q crashed or partitioned after q received m, but before r and s did. This may have undesirable effects, if processes have stable storage or partitioning may occur: when q recovers or merges, its state S_m diverges from the state of r and s, S_k.

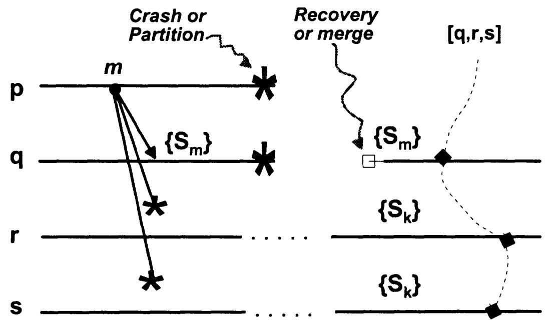


Figure 7.4. Non-Uniformity

You may ask yourself why should someone bother with these subtle differences instead of just using the stronger form of consensus in all cases. The issue is that protocols that solve non-uniform consensus can be more efficient. Consider for instance the following protocol that relies on perfect failure detectors.

As in the previous section, we assume that there is a total order on process identifiers and that the process with the smallest identifier is the coordinator for consensus. The coordinator sends its value to every process in the system and, when this value is received, a process decides immediately. If the coordinator remains correct, eventually every process receives and decides the same value. If the coordinator crashes, the next process in the line assumes the role of the coordinator. The new coordinator asks every other correct process if they have decided. If at least one correct process has decided, the new coordinator forwards the value that has been previously decided to every other correct process. Otherwise, if no decided value is known by the processes that remain correct, the coordinator decides and disseminates its own initial value (note that the previous coordinator as well as other crashed processes may have decided differently).

Now compare the protocol described above with the protocol that we have presented in the previous section. While in the non-uniform version a process may decide as soon as it receives a proposal from the coordinator, in the uniform version, this proposal has to be acknowledged by (at least) a majority of correct processes before being decided. Thus, in all applications where the state/actions of crashed processes cannot compromise the correctness of the system, the non-uniform version of the protocol provides much better performance.

7.4 MEMBERSHIP

We have already seen many examples of distributed activities where several processes cooperate to achieve a common goal. We can refer to the set of collaborating processes as a *process group*, which has a membership. We have introduced the notion of group membership under *Consistency*, in Chapter 2. Here, we review the problem and analyze the requirements of membership protocols in the presence of failures and/or partitioning.

The membership of the group is the set of processes belonging to the group at a given point in time. A *membership service* keeps track of the group membership and provides this information to the group members in the form of a *group view*, the subset of members that are mutually reachable at a given point. The group membership is often dynamic: in response to user demand or changes in the runtime environment (load, failures, etc) the group can grow, by letting new processes *join* the group, or shrink, by letting members *leave* the group. Processes may also become involuntarily unreachable because of failures.

7.4.1 Group Membership

At first glance, the task of providing participants with information about who belongs to the group may seem rather simple. However, group membership is as a form of distributed agreement (all group members must agree on what is the group view) and we know distributed agreement in the presence of faults is an intrinsically complex problem. Actually, it turns out that even *defining* the properties of a group membership service is a difficult task, and several different types of membership service have been proposed in the literature. We will discuss the different alternatives in the next few paragraphs. For now, let us just state the informal and intuitive definition of consistent membership information: if the membership of the group remains unchanged and there are no link failures, all members should obtain the same group view.

Usually, the agreement service is required to remove processes that have failed from the group view. So the membership information needs to be consistent and *accurate*. However, as we have seen before, accurate failure detection can be hard or even impossible to perform. In this case, how should the membership service behave? Assume that a group member p is suspected (it shows no activity). If the membership service does not remove p from the group, the application trusts p to work properly. For instance, the application might be using a work distribution algorithm where every process contributes with its share of work. While p remains in the view, other processes will expect it to do its job.

On the other hand, if the process is removed from the group it will not participate in the progress of the application and will eventually become desynchronized with respect to the other (active) processes. In practice it will become unable to contribute to the group unless some resynchronization procedure is executed to update its state. Recovery is something we will discuss later in this chapter.

The order by which the information is provided to the users is also relevant. Application code can be made simpler if changes in the group membership are received in the same order by all members. For instance, if a process is notified that p has failed, and later that q has also failed, then all other correct processes should be notified of the failure of p and q in that same order. Note that membership heavily relies on failure detection. Inaccurate failure detection may cause membership to have erratic behavior.

7.4.2 Linear Membership

A linear membership service is characterized by enforcing a total order on all views, i.e., all correct processes receive exactly the same sequence of views, as illustrated in Figure 7.5. As with the consensus problem, the linear membership service can be uniform or non-uniform. If it is defined as uniform, the history of views delivered to a crashed processes is necessarily a prefix of the history of views delivered to the correct processes.

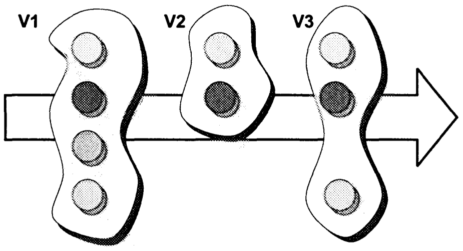


Figure 7.5. Linear Membership

In a synchronous system without partitions, linear membership can be easily enforced. Processes are either correct or crashed. All correct processes can detect the crashes and can communicate among themselves. Membership just requires agreement on the content of each view.

In a system subject to network partitions, or in an asynchronous system where failure detection is unreliable, ensuring a linear membership is more difficult. For instance, if the system is split in two non-communicating partitions, only one of these partitions, called the primary, may continue to deliver views. Because of this constraint, the linear membership is also called *primary-partition* membership.

What happens to the processes that are not in the primary partition? Three alternatives are possible. One is to block these processes while the partition persists and let them catch-up with the others as soon as the partition is healed. The other is to force these process to crash. Later they can be activated as new processes and join the system again. The third alternative is to implement a partial membership service, as described next.

7.4.3 Partial Membership

The primary-partition model is quite intuitive but too restrictive for certain type of applications. It is often interesting to keep delivering views in both partitions. Both sides continue to operate and when the partition is healed, the state is reconciled (in a manner that is usually application-dependent). Thus the group splits and merges in response to changes in the network connectivity. Views are no longer totally ordered, instead a *partial* order of views is provided. It is possible to define different types of partial membership according to the amount of overlap that is allowed among views delivered in different partitions. Of all the possible definitions of partial membership, the *strong* partial, illustrated in Figure 7.6, is the most intuitive. According to this model, concurrent views never intersect. In other words, V_2 and V_2' correspond to two completely disjoint partitions, which later merge again into V_3. Strong partial membership supports the virtual synchrony paradigm defined in Chapter 2 (*see Consistency*).

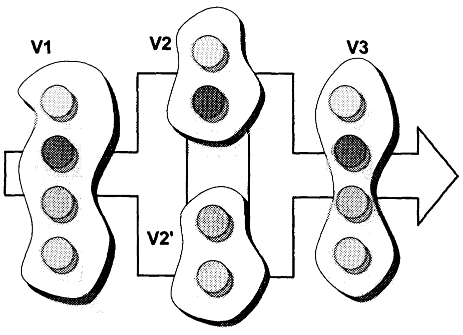


Figure 7.6. Strong Partial Membership

7.5 FAULT-TOLERANT COMMUNICATION

Fault-tolerant communication is about making sure that two or more processes can exchange information despite the occurrence of failures in the communication link or in some of the participating processes. As we have seen before, the major types of failures that need to be handled are: timing, omission, and value failures (message corruption). Malicious links can also spontaneously generate messages. In this section we discuss how these failures can be addressed, both for point-to-point and for multicast communication.

7.5.1 Reliable Delivery

We will start our study with omissions. From the previous chapter, the reader should already have a good idea of how communication can be made reliable. Two main alternatives are available: error masking or error recovery, represented in Figure 7.7. Error masking can be based on spatial or temporal redundancy. Spatial redundancy consists in deploying several links connecting the communicating processes (Figure 7.7a). In order to mask k omissions, $k+1$ links should be used. Of course, each message should be sent through all the links. Temporal redundancy consists in sending the same message several times. Again, in order to mask k omissions, the message should be sent $k+1$ times (Figure 7.7b). In both cases, duplicates should be discarded at the recipient. Error masking is appropriate when the assumed number of successive omissions k, also called the *omission degree*, is small and the need for fast recovery compensates the waste of bandwidth.

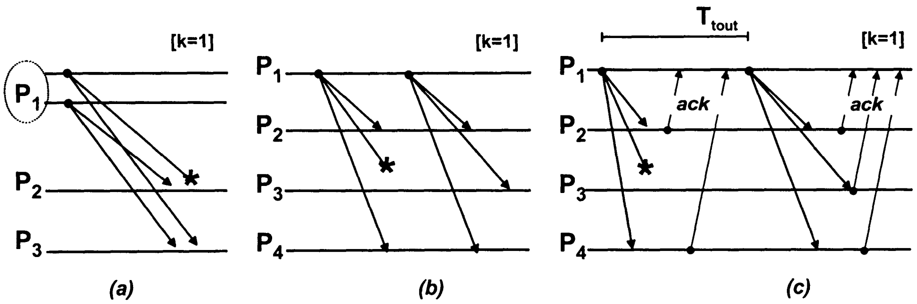


Figure 7.7. Communication Error Processing: (a) Masking (Spatial); (b) Masking (Temporal); (c) Detection/Recovery

Error detection and recovery is based on acknowledgments and timeouts. Acknowledgments can be sent whenever a message is received (*positive acknowledgment*, as in Figure 7.7c) or only when the loss of a message is detected by the recipient (*negative acknowledgment*). In the positive acknowledgment scheme, a message is retransmitted if a confirmation is not received by the sender before a timeout occurs. In the negative acknowledgement scheme, a retransmission is requested by the recipient by sending a negative acknowledgment back to the sender. The former method offers a faster failure detection with sporadic traffic. The latter minimizes network traffic but requires one of two things: (a) a stream of numbered messages from the sender to the recipient, such that the recipient detects the lack of a message if it receives message i but not message $i - 1$; or (b) a time-triggered lattice such that the recipient knows it should receive a message at a given time. Error detection and recovery offers a better usage of network resources (in particular, the negative acknowledgment scheme) since retransmissions just happen when an omission actually occurs (and in most modern networks, with the notable exception of wireless communication, omissions are relatively rare).

In most practical systems, a given message is not retransmitted an infinite number of times: after a pre-defined number of retransmissions, the communication is aborted in the assumption that either the recipient or the link have crashed. This is equivalent to implicitly assuming that the system has some degree of synchrony, but this assumption is not always substantiated. The necessary steps to implement error/failure detection this way are embodied in a technique called *bounded omission degree* (*see Real-Time Communication Models* in Chapter 13).

7.5.2 Resilience to Sender Failure

We have just seen how to deal with network omissions. We now discuss the problem of sender failure. In the case of point-to-point communication this is relatively easy since it basically is a problem of failure detection. Note that if the sender crashes before successfully transmitting the message, the message is lost. Failure detection comes into play just to prevent the receiver from waiting forever for the missing message.

In multicast communication the failure of the sender is more problematic. If there are omissions in the link it is possible for a message to be received just by a subset of the participants. Usually, the sender of the message has the responsibility for retransmitting it. In such case, the failure of the sender may leave the system in an inconsistent state. At this point, it is worth to distinguish three levels of reliability in multicast, illustrated in Figure 7.8:

1. *Unreliable multicast.* The weakest form of multicast, where no effort is made to overcome link failures. Basically, multicast is as reliable as the link and the sender are (if one of these components exhibits a fault, some or all recipients may lose the message).

2. *Best-effort multicast.* A multicast where the sender takes some steps to ensure the delivery of the message, like retrying or repeating. However, if the sender fails, no reliability can be guaranteed.

3. *Reliable multicast.* A multicast where the participants coordinate to ensure that the message is delivered to all correct recipients (as long as it is delivered to at least one correct recipient). For instance, a recipient takes over a failed

sender. As with consensus, there is a stronger definition of reliable multicast, *uniform multicast* that we will discuss later in this section.

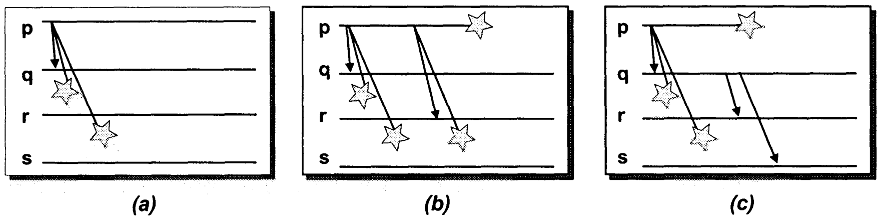


Figure 7.8. Multicast Reliability: (a) Unreliable; (b) Best-effort; (c) Reliable

How can reliable multicast be obtained? Well, we are back to error masking and error recovery. In an error masking approach, all recipients retransmit a message as soon as they receive it. Thus, even if the sender fails, each recipient makes sure the every other recipient also receives the message. If an error recovery approach is used, recipients keep a copy of the message but only retransmit the message when the failure of the sender is detected. Of course, message copies cannot be kept forever since they consume precious memory resources. Thus, some mechanism must detect when a message has been delivered to all intended recipients such that its copies can be discarded. Such a mechanism is usually called a *stability tracking protocol.*

The error masking approach is more appropriate for systems where accurate failure detection is impossible (for instance, asynchronous systems). However, unless channels are assumed to be reliable (an abstraction), the message has to be retransmitted forever. As we have noted, most systems give up after a certain number of retransmissions, which implicitly means the failure of the recipient. Error recovery also assumes that failures can be detected. As a matter of fact, the multicast reliability is strongly related with the issue of membership (failure detection and notification).

7.5.3 Tolerating Value Faults

So far, we have been discussing how to tolerate omission faults. However, messages can also be corrupted during the transmission, so it is also necessary to tolerate value faults. Fortunately, most of these faults can be detected using checksums or signatures. Messages corrupted are then simply discarded, and the value fault transformed into an omission fault, which we know how to handle.

Checksums do not cope with all the possible value faults. Value faults can be caused by a faulty sender that produces an error before the checksum is computed. Semantic faults of this type can only be tolerated using space redundancy, i.e., by comparing the values produced by different sources of the same logical value. If we have several recipients, it is important to ensure that, for consistency, the result of the comparison is the same at all correct processes.

In other words, we need to ensure that correct recipients can agree on the outcome of the comparison. To achieve such goal, a consensus algorithm must be executed among the recipients.

7.5.4 Tolerating Arbitrary Faults

Arbitrary faults are harder to tolerate. A faulty sender may send conflicting information to different recipients. Also, if the link exhibits malicious behavior, it can spontaneously generate a message that is syntactically correct, impersonating a legitimate sender. The problem of reaching consensus in the presence of arbitrary faults is called *Byzantine Agreement* after a paper by Lamport, Shostak, and Pease (Lamport et al., 1982).

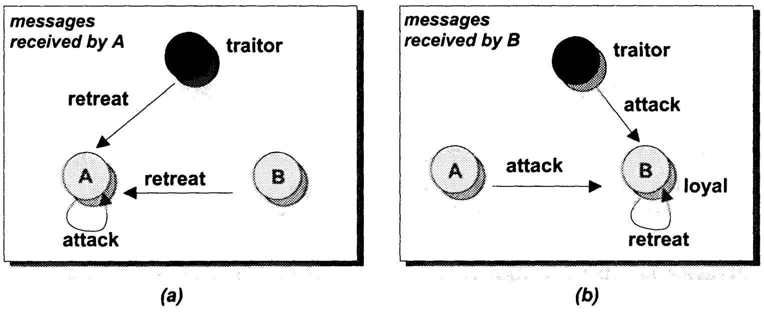


Figure 7.9. Traitor Sends Conflicting Information

The Byzantine Agreement problem can be formulated as follows. A number of generals, in face of an enemy army, must decide whether to attack or to retreat. Most of these generals are loyal to each other (correct) but some are traitors (faulty). In the presence of favorable conditions, the combined force of the loyal armies can defeat the enemy. However, unless all loyal generals attack together, their troops will be defeated. The problem is that loyal generals must agree on a single binary value (attack/retreat) despite the presence of traitors that will, maliciously, try to prevent agreement from being reached.

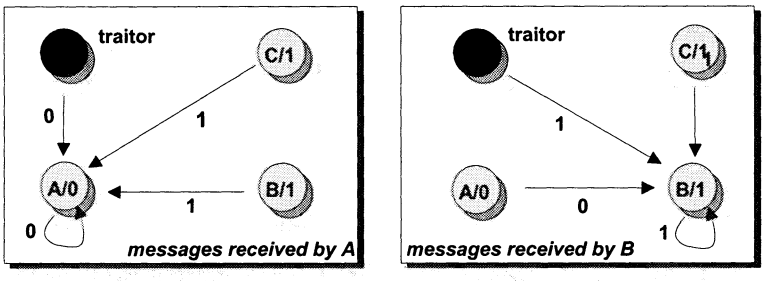


Figure 7.10. One Round of Byzantine Agreement (attack:1; retreat:0)

Assume that the system is synchronous and that the agreement protocol operates in rounds. In each round, generals send messages to every other general. However, traitors are free to omit some or all messages and to send conflicting messages to different generals. The initial value proposed by each loyal general consists of his own assessment of the correct decision: to attack or retreat. How many traitors are sufficient to prevent agreement? Consider a scenario with three generals, two of them loyal as illustrated in Figure 7.9. Given the intuitive notion of majority vote, a correct decision should be possible. We will see that it is not so. One of the loyal generals, A, wishes to attack, whereas the other, B, believes that the armies must retreat. Assume that A receives two retreat messages, one from B and one from the traitor (Figure 7.9a). Since there is a majority of retreat messages can A safely decide? Unfortunately, the traitor can send a conflicting message to B supporting A's proposal to attack (Figure 7.9b). A simple majority would force A to retreat and B to attack! Another interesting finding is that additional rounds of message exchange do not provide any help. In fact, it can be proven that at least $3f + 1$ processes are needed to tolerate f Byzantine faults so, in our previous example, it is impossible to ensure that loyal generals always reach agreement.

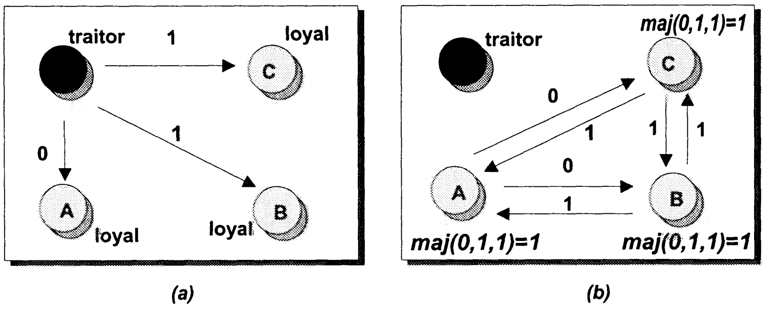


Figure 7.11. (a) First and (b) Second Rounds of Byzantine Agreement (partial view, faulty sender)

What happens if we add an additional loyal general to the system? Figure 7.10 shows a scenario after the first round of messages. Assume that the loyal generals have pre-agreed that they should follow the majority and, in case of ties, retreat. As the figure shows, consensus cannot be reached in one round of messages, even with more processes in the system. In the example, two loyal generals (B and C) want to attack and another loyal general (A) wants to retreat (from now on, we replace 'retreat' by 0 and 'attack' by 1 to simplify the figures). As before, by sending an attack vote to B and a retreat vote to A, the traitor can force A and B to disagree. However, we now have enough redundancy in the system to mask the influence of the traitor with an additional round of messages. For each sender p, the other three remaining processes exchange the values they have received from p to agree on the value sent by p. Let us see why, in this case, an additional round of messages (for each value proposed) solves the problem. There are two cases:

- *The sender of the value is faulty.* In this case p can send inconsistent votes to the loyal processes, as illustrated in Figure 7.11. However, since all the remaining generals are loyal, after exchanging among them the value received from the traitor they have exactly the same set of values. If they use a majority criteria, all correct processes will select the same final value.
- *The sender is correct.* In this case, p disseminates exactly the same value to all processes, as illustrated in Figure 7.12. When the remaining three processes exchange the value received, the two correct processes forward the value originally sent by p and the faulty process forwards some arbitrary value. Still, since there is a majority of correct values, the value originally sent by p will be chosen.

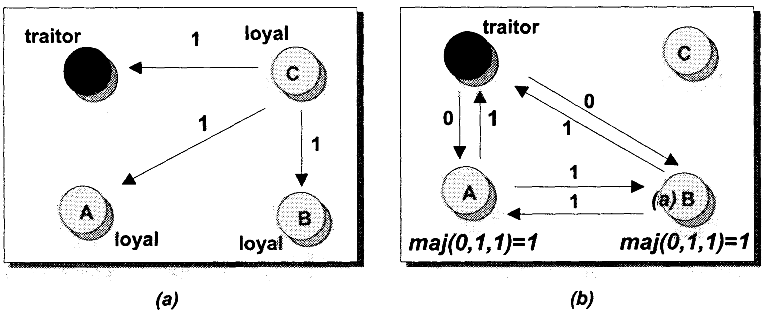


Figure 7.12. (a) First and (b) Second Rounds of Byzantine Agreement (partial view, correct sender)

Since after this second round of message exchanges every correct process has obtained exactly the same input from every other process in the system, correct processes can use a majority function to select the final decision of the Byzantine agreement. The interesting feature of this solution is that in order for the correct processes to agree on a common decision, they have first to agree on the input value provided by each of the processes in the system (correct or faulty). This recursive approach works because in the second round there is one less degree of freedom in the system.

Actually, this recursive approach can be extended to tolerate additional faulty processes. Let us call the complete protocol described above *Byzantine(1)* and the protocol executed in the second round *Byzantine(0)*. The protocol *Byzantine(1)* can be described as follows: first, for each process p, the remaining processes execute *Byzantine(0)* to agree on the value proposed by p, then they use a majority function on the agreed values to select the final decision. A protocol *Byzantine(f)* to tolerate f faulty processes (in a system with at least $3f+1$ processes) could be described using the same recursion: first, for each process p, the remaining processes execute *Byzantine(f-1)* to agree on the value proposed by p, then they use a majority function on the agreed values to select the final value.

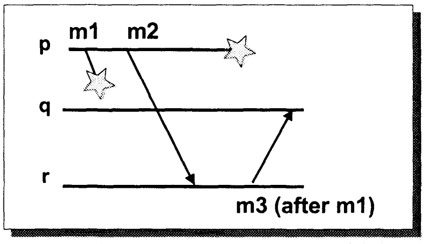


Figure 7.13. Causal Hole

7.5.5 *Securing Causal Order*

In Part I of this book, we have discussed a number of ways to enforce causal order (*see Ordering* in Chapter 2). Crash faults can affect these protocols in different ways. Consider the following example, illustrated by Figure 7.13. Process p sends a point-to-point message m_1 to process q, immediately after it sends another message m_2 to process r and then crashes. Message m_1 is lost but m_2 is received. Assume that there are no other messages involved. Can r deliver m_2? If local criteria are used, the answer is yes. Although $m_1 \rightarrow m_2$, m_1 was not sent to r, thus delivering m_2 does not violates causal order. However, if r delivers m_2 it will be *contaminated*: its state will causally depend on a message that cannot be recovered. A *hole* in the causal history will be created. Any subsequent message from r to q will not be delivered (it causally depends on m_1, which cannot be recovered).

To avoid contamination, before delivering m_2, a process must be sure that all preceding messages have been delivered or, at least, that there are enough copies of the message in the system to ensure its future delivery. Note that if $k + 1$ processes have a copy of the message, the message can be recovered even if k processes fail. Also note that this is the default behavior of a causal protocol when all the messages are sent to the same set of processes. Stability tracking protocols are useful to reduce the history to be kept.

7.5.6 *Securing Total Order*

Enforcing total order requires some form of agreement among the participating processes. In fact, the problem can be expressed in terms of having the processes agree by which order the messages should be delivered. As in the consensus problem, non-uniform and uniform versions of total order can be defined. There is more than similarity between these two concepts. In fact, it has been shown that uniform consensus and uniform atomic broadcast are equivalent problems in different classes of distributed systems. In other words, you can build a

```
Sender p:
    Initialization
        let S be the sequencer;
    procedure multicast(m) begin
        send UNORDERED(m) to S

Sequencer S:
    Initialization
        let sn := 0; // sequence number
    upon receive UNORDERED(m) from p
        sn := sn+1;
        send ORDERED(sn, m) to all recipients

Recipient q:
    upon receive ORDERED(sn, m) from S
        deliver m;
```

Figure 7.14. Simple sequencer based total order algorithm

fault-tolerant total order primitive based on a uniform consensus primitive but you can also build a consensus primitive based on total order.

This last reduction is actually very simple: all processes atomically broadcast their values and all processes agree on the first value delivered by the underlying atomic broadcast layer. Alternatively, a deterministic function is applied to the set of all process values, in order to extract the same value everywhere.

We have already discussed the challenges posed by failures in the agreement problem. These difficulties also apply to the total order protocols. Consider for instance the simple sequencer-based algorithm to enforce total order whose pseudo-code is shown in Figure 7.14. We recall that, in its simpler form, all processes send their messages to a centralized sequencer site, which then forwards them to all recipients. The delivery order is defined as the order by which messages are sent by the sequencer. When the sequencer fails, a new sequencer must be elected. However, some messages may have been received by some processes but not by others. For instance, in Figure 7.15, messages $m1$ and $m4$ are delivered to q and r but not to p. The new sequencer has to retransmit these messages in the same order to p, otherwise the system will be contaminated. In order to obtain ordering information, the elected sequencer must gather information about which messages have been delivered and in which order (assume that p would be elected as the new sequencer— it could learn about $m1$ and $m4$ through q and r). Since crashed processes cannot be inquired, the order established by the new sequencer may be different from the order of delivery in processes that have crashed. This also means that a simple sequencer algorithm cannot ensure uniform total order.

Enforcing total order using consensus as a building block can be done as illustrated in Figure 7.16 (we use the algorithm presented in (Chandra and Toueg, 1996)). Messages are disseminated using an unordered primitive guaranteeing that all correct processes will eventually receive all messages sent by

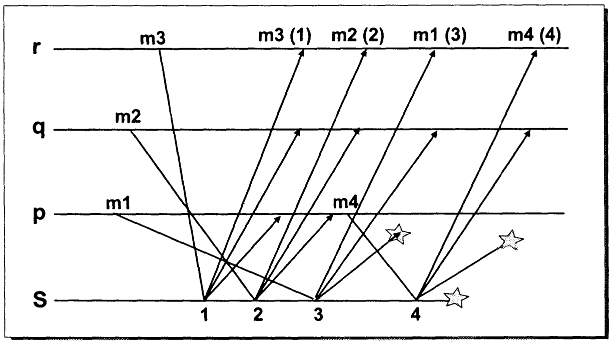


Figure 7.15. Sequencer-Based Total Order

```
Sender p:
    Initialization
        unordered ← ∅ ;
        ordered ← ∅ ;
        k ← 0 ;
    procedure total-order-broadcast(m) begin
        broadcast UNORDERED(m)

    when receive UNORDERED(m) from p
        unordered ← unordered ∪{m}
    when (unordered-ordered) ≠ ∅ begin
        k ← k + 1 ;
        propose (k, unordered-ordered);
        wait until decide (k, msg_set);
        ordered ← ordered ∪ msg_set; // deliver msg_set in some deterministic order
```

Figure 7.16. Chandra and Toueg's total order algorithm

correct processes (this can be obtained by letting each recipient forward each message to all other processes). Messages received this way are saved in a bag of unordered messages. To order messages, each process executes a sequence of consensus, each consensus orders a set of messages. For consensus number i, a process p proposes its bag of unordered messages. The result of the i^{th} consensus is the set of messages which all correct processes agree to assign sequence number i to. These messages are removed from the bag of unordered messages and delivered to the user according to some deterministic rule before a new round of consensus is started.

The fault tolerance aspects of some previously studied ordering protocols (*see Ordering* in Chapter 2) also worthwhile mentioning. The token-site total ordering method survives f failures by rotating the token and copying relevant ordering information $f + 1$ times, before considering a message stable. In Δ-protocols ordering information can be evaluated locally at every recipient. So

FT concentrates on two aspects: ensuring timely delivery; making clock synchronization fault-tolerant. Exceeding the assumed delivery delay or precision bounds can make the protocol fall apart.

The reader should also note that protocols providing nonuniform total order may also cause the contamination of the system. Consider for instance a sequencer-based total order algorithm. The sequencer receives two multicasts m_1 and m_2 and delivers them by that order. It subsequently multicasts a new message m_3 whose contents depends on the delivery order of m_1 and m_2. The sequencer then crashes before forwarding the delivery order to the remaining processes. The surviving processes receive m_1, m_2 and m_3 and decide to deliver those messages. In a nonuniform algorithm, the new sequencer is free to assign any order to these messages, for instance: $m_2 < m_1 < m_3$. Unfortunately, the contents of m_3 is now inconsistent with the selected delivery order. A uniform total order protocol prevents this case from occurring (but at a greater cost).

7.6 REPLICATION MANAGEMENT IN PARTITION-FREE NETWORKS

In the next few sections we will address a basic technique to provide continuity of service and/or availability of data: spatial redundancy in the form of replication. We begin by making strong assumptions about the system: the network is not subjected to partitions and the processes only fail by crashing.

7.6.1 State Machine

Before proceeding, it is useful to distinguish the behavior of components from the determinism point-of-view. The state and outputs of a *deterministic* component depend exclusively on its initial state and on the history of *commands* that it has processed. A deterministic component, also called a *state machine*, has been characterized as follows (Schneider, 1993):

> **Semantic Characterization of a State Machine**– Outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in the system.

Figure 7.17a depicts the principle of the state machine. If two components, executing the same state machine: are started with the same initial state; and execute exactly the same (totally ordered) sequence of commands; then they always exhibit the same behavior (as long as they remain non-faulty). It is also useful to distinguish *write* commands, that cause the state of the component to change, from *read* commands, that do not cause a state update.

On the other hand, the state and behavior of a *non-deterministic* component depend not only on the sequence of commands it executes but also on local parameters that cannot be controlled. Unfortunately, there are many mechanisms that can cause a non-deterministic behavior: non-deterministic constructs in programming languages such as the Ada select statement; scheduling decisions; resource sharing with other processes; readings from clocks or random number generators; etc. The state of two non-deterministic replicas is likely to diverge even when they execute same sequence of inputs.

Between these two extremes it is also useful to define *piecewise deterministic components* (Strom and Yemini, 1985). The execution of these components can be decomposed into several deterministic sequences of instructions, intertwined with non-deterministic steps, such as the processing of a sporadic event or message. A state machine where command processing order depends on non-deterministic events such as urgent message arrivals or internal scheduling decisions is a piecewise deterministic component, where the granularity of the sequence is the command.

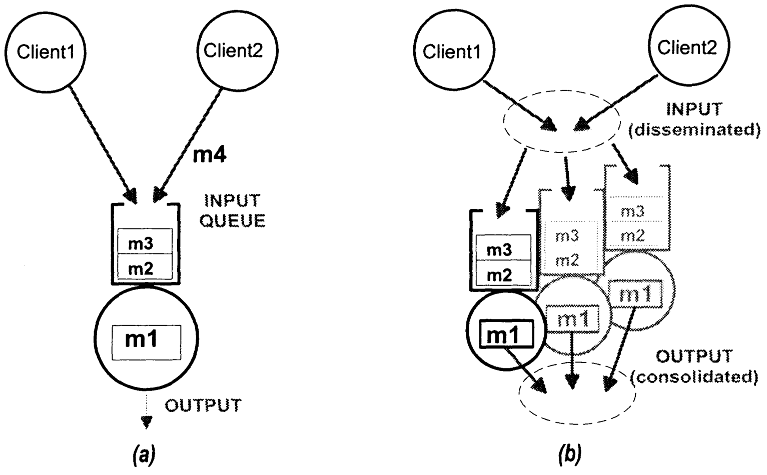


Figure 7.17. State Machine: (a) Simplex (b) Active Replication

7.6.2 Active Replication

Active replication is an intuitive technique that can be applied to state machines. It consists of having several replicas of the same state machine executed by different processes. In order to ensure that the state of these replicas is kept consistent, an atomic multicast protocol must be used to disseminate the state machine commands. The atomic multicast primitive ensures that all replicas receive the same commands in the same order, as illustrated in Figure 7.17b. When the state machine produces an output, all replicas produce the same result. The consolidation is simple in an omissive fault model: any of the individual results can be used (remember, no value faults yet).

When active replication is used, replica consistency is preserved implicitly by the atomic multicast protocol used to distribute commands. No explicit recovery procedure is required when one of the replicas fails: the remaining replicas will continue to provide service. Of course, the approach has its disadvantages. It is resource demanding, because each replica requires a full set of resources to operate. Also, it requires the use of total order, which is intrinsically less efficient than weaker communication primitives. With regard to this last point, it is worth mentioning that only write commands need to be totally ordered: read commands can simply be causally ordered.

7.6.3 Semi-Active Replication

Active replication can only by applied to state machines. Semi-active replication is a variant of active replication that can also be applied to piecewise-deterministic components. Also called *leader-follower*, the idea is that all replicas execute the commands but a single replica (the leader) is responsible for making all non-deterministic scheduling decisions and provide this information to the other replicas (the followers) as illustrated in Figure 7.18a.

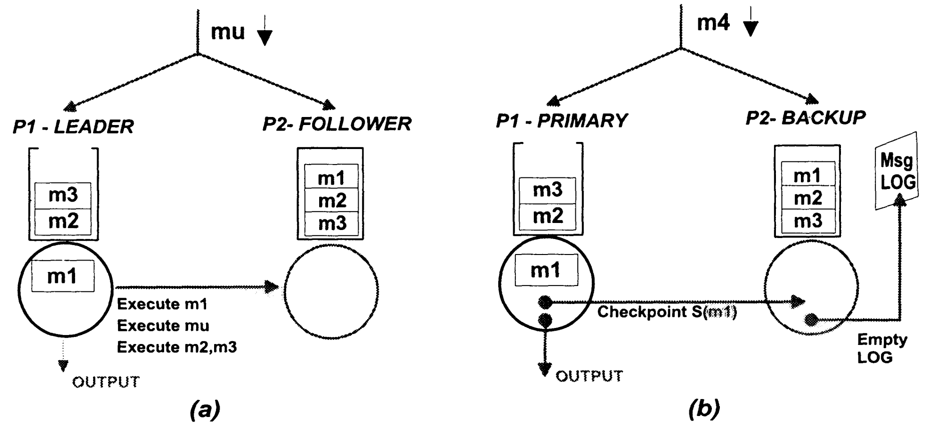


Figure 7.18. (a) Semi-Active Replication; (b) Passive Replication

Since the order of execution is defined by the leader, this technique does not require the use of totally ordered broadcast. Unordered unreliable broadcast can be used instead: note that the queues of the two replicas have the same messages, in different orders. Follower replicas log all messages but delay the processing until the leader sends them the result of its non-deterministic decisions, namely the execution order. For example, note that the leader sends instruction $\langle execute m_1 \rangle$, then an urgent message m_u arrives and preempts the foreseen execution sequence (m_2, m_3). The next instructions from the leader will thus be to execute m_u, and then m_2, m_3. To avoid inconsistency, every non-deterministic decision taken by the leader must be disseminated immediately to the followers. So there is a tradeoff among the degree of consistency, the degree of non-determinism, and the cost of leader-follower synchronization.

7.6.4 Passive Replication

Both active and semi-active replication require the replicas to execute commands, and this doubles the resource requirements. Passive replication is more economic, since only one replica (called the *primary*) executes commands. The other replicas, called *secondary* or *backup*, remain in idle state, although they receive and log all commands. Periodically, the primary replica *checkpoints* its state in a location that can be read by the backups: it can send its state directly to to the backup replica(s) (using messages, as shown in in Figure 7.18b) or save it on a shared repository. All backup replicas clear their logs after a checkpoint. When the primary fails, one of the backup replicas is elected as a new primary and resumes operation from the last *checkpoint*, executing from the log to catch-up with the state of the primary just before failing. Depending upon the size of the replica state and upon the frequency of checkpointing, each checkpoint can include the complete state of the replica or just the updates from the last checkpoint (*incremental checkpointing*). The reader should note that there is room for tradeoffs between: size of checkpoint, size of log, frequency of checkpoints, recovery glitch.

7.6.5 Lazy Replication

Lazy replication (Ladin et al., 1992) can be seen as a hybrid scheme that mixes active and semi-active replication. The approach uses semantic knowledge to distinguish the operations that need to be executed with total order from those that can be executed in different orders in different replicas.

Clients forward their request to one of the replicas. If the request is an update that needs to be totally ordered, the replica synchronizes with the other replicas to establish a total order for that request; all replicas will apply the update in the same order and the contacted replica will reply to the client. The reply carries a vector clock that must be returned by the client in subsequent requests (this ensures that the client sees the updates in an order consistent with causality, even if it contacts different replicas). Requests that do not need to be totally ordered are executed in an order that respects causality by the replica that receives the request. A reply is sent to the client and the request is forwarded to the other replicas in background (thus, the name of lazy replication).

7.7 REPLICATION MANAGEMENT IN PARTITIONABLE NETWORKS

All of the replication schemes just discussed assume reliable failure detection. Besides, replica divergence is not avoided in the case of network partitioning. We will now present replication schemes that ensure consistent service even when some replicas become mutually unreachable. These techniques work in networks where partitions can occur and in systems where replicas can crash and later recover.

7.7.1 Static Voting

The idea behind voting is that any given operation should only be allowed to proceed if a minimum *quorum* of replicas, or copies, can perform it. Quorums must be defined in such a way that conflicting operations always intersect in at least one replica. This common replica is able to make the outcome of the previous operation available to the replicas executing the new operation. The most current state can be identified by having each replica maintain a *version number* that is incremented every time the data are updated.

The simplest example of a quorum algorithm for managing replicated data is one where read operations are allowed to read any single copy, and write operations are required to write all replicas of the object. This *read-one write-all* algorithm provides read operations with a high degree of availability at a very low cost. On the other hand, it severely restricts the availability of write operations since they cannot be executed after the failure of a single copy.

Weighted voting An extension to the above-mentioned scheme consists in assigning each copy a number of *votes*. Quorums are defined based on the number of votes instead of the number of replicas and the condition that guarantees overlapping consists in requiring that the sum of *quorums* for conflicting operations on an item should exceed the total number of votes for that item. This technique is the basis for a set of algorithms named *majority voting* (Thomas, 1979) and *weighted voting* (Gifford, 1979). Weighted voting is based on the following: given n the total number of votes for an item, and r and w the quorums required for read and write operations respectively, then it should be $2w > n$, and $w + r > n$. The intuition behind this can be explained by an example. Suppose $n = 7$, $r = 4$ and $w = 4$. Then, if a partition containing replicas summing at least 4 votes is written, only 3 votes are left, not enough to write divergently in other partitions (first condition). The second condition is similar, ensuring that one read and one write cannot be made concurrently (in two partitions), but are serialized regardless of how partitions develop. Namely, if the write occurs first, then the read is sure to include at least one of the replicas that have seen the previous write. This replica can update the others, ensuring sequential consistency of the history of operations. The separation between number of replicas and number of votes is the key point: a careful vote assignment taking into account the properties of each individual replica may yield improved results on the availability of the system. In the example above, suppose that replica A stored in a node with better reliability and/or connectivity is given 3 votes, and all other four replicas are given 1 vote: a partition with A and any other replica secures the majority of votes, allowing system operation to proceed even if a majority of replicas fail or partition.

Coteries An alternative approach to describing quorums, in particular weighted quorums, is to use an explicit set of processes, or *quorum groups*. The collection of quorum groups used by an operation is called the *quorum set*. To ensure overlapping, each group in the quorum set must overlap with every other group in that set. A quorum set with such characteristics is called a *coterie* (Garcia-Molina and Barbara, 1985) and it can be used to achieve mutual exclusion. The reason for explicitly invoking an operation on a quorum group is that there are quorum sets which cannot be defined by voting algorithms. Although quorum sets are the most generic way to describe coteries, they are expensive: all quorum groups must be stored locally, and a search for the quorum group must be performed upon every operation. This is significantly more expensive than voting, where only the local vote and quorum need to be stored (and checked) at each process. Additionally, the number of quorum set alternatives is so vast that it is difficult, if not impossible, to choose the most appropriate quorum set for a given system. Due to this reason, other representations of quorum sets that are comparable to coteries have been searched.

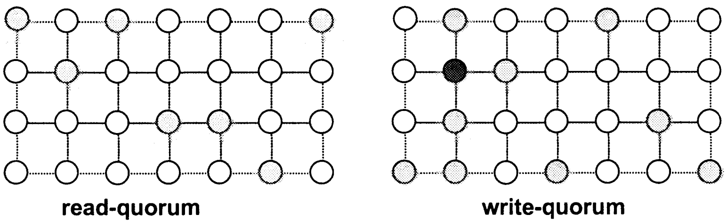


Figure 7.19. Grid Read and Write Quorums

Structural representations One of the methods to describe quorum sets is to use logical structures such as a tree, a grid, etc. Let us give some examples. The *tree quorum* algorithm (Agrawal and El-Abbadi, 1991) organizes replicas in a logical binary tree. The algorithm tries to find a quorum group by selecting a path from the root of the tree to any one leaf. If no such path can be found due to the inaccessibility of a copy c, that copy must be replaced by two paths, both starting at children of c (the algorithm is recursive). One problem with this algorithm is the probability of overloading the root node, since it belongs to all (failure-free) quorums. Another example is the *grid algorithm* (Cheung et al., 1990), that organizes the replicas in a logical rectangular grid: a read quorum must contain a node from each column and a write quorum must contain a read quorum and all the nodes in a column of the grid. In this way conflicting operations are guaranteed to overlap, as shown in Figure 7.19.

Byzantine Quorum Systems The systems described above are designed to tolerate benign faults such as crashes or partitions. Thus, the intersection of quorums for conflicting operations is required to have at least one process. It is possible to tolerate additional types of faults by using larger intersections. For instance, quorum systems that are able to mask Byzantine faults can be constructed by ensuring that quorums contain a majority of correct processes (Malkhi and Reiter, 1998). For instance, in a read operation, a client can accept a value that is returned by at least $f + 1$ servers (and ignore arbitrary values returned by faulty servers).

7.7.2 Dynamic Voting

In the algorithms described above, quorum sets are chosen *a priori* at system design time. At run time, a node simply determines the reachable set of sites before performing an operation, and checks if this group belongs to the quorum set designated for the operation. In contrast, in a dynamic scheme, quorum groups can be changed during runtime. The idea is to use information about the current system configuration (such as which nodes are down) to adapt the algorithm in order to maximize some performance criteria. Typically, dynamic schemes attempt to make the system operate with small read quorums, since read operations are usually predominant. Consequently, the corresponding write quorums require a larger number of replicas and may become hard to reach when failures occur. In this case, after a failure the system should switch to another, more favorable, read and write quorum sets.

Most of the algorithms described in the previous section can be extended to take reachability information into account. Dynamic variants of voting algorithms are based on similar principles: partitions are identified in some form and updates are performed on replicas that are in the same partition. For instance, the weighted dynamic voting algorithm (Davcev, 1989) changes the majority criteria whenever a network partition is suspected. Whenever this criterion is changed, the version number of each copy at that point is stored (this information is called a *partition vector*). This allows to keep track of which replicas have participated in the most recent majority; the majority criteria can only be changed when a majority of up-to-date replicas are reachable. When communication is re-established, partitions are merged using the partition vector information. Usually, to avoid degrading the performance of read operations, a new partition is only installed during write operations (Agrawal and El-Abbadi, 1990).

7.8 RESILIENCE

The degree of resilience assumes at least two facets. The first is qualitative, and concerns the kind of faults to be tolerated, for example: whether or not the system can partition; or whether time- or value-domain faults are assumed. The second is quantitative, concerning the number of faults to be tolerated.

7.8.1 When to Compare Results?

Voting and comparison are common activities in fault tolerance paradigms. In the last section, we studied the use of voting to assure a tradeoff between consistency and availability: assessing the existence of progress conditions, such as a quorum or a majority of replicas or votes, in order to let a computation proceed without the risk of inconsistency.

In order to tolerate value faults, different sources of the same "logical" value must be available in the system, so that their values can be compared, or voted upon. Of all the techniques discussed so far, only those involving space redundancy, such as active replication, are able to provide tolerance of value faults, since the outputs are computed by each replica with complete independence from the other replicas. The correct sources will produce a valid value and the faulty sources an incorrect value. Voting on these results yields the correct value by masking the error, or at least allows detecting the error, depending on the amount of redundancy.

Voting is very simple when all correct values must be exactly the same and can be compared bitwise. For example, in the case of a single assumed fault and given a vector of values to be compared, a two-element vector (two replicas) allows detecting an error when the two entries are different, but there is no recovery, since it is a no-winner situation. A three-element vector allows masking one error, by deciding for a majority (at least two) of equal values.

7.8.2 Exact and Inexact Agreement

We have discussed how to pick a correct value from a set of values produced by different replicas. In many fault-tolerant architectures, the consumer of the value is also replicated, and distributed. This means that, in order to preserve consistency, all consumer replicas need to select not only a correct value but also the same correct value. In other words, the consumer replicas need to *agree* on the correct value. In omissive failure systems, if all producer replicas atomically broadcast their values, consumer replicas end up with the same vector of producer replica values, and thus can do a deterministic comparison, arriving at the same value. As a matter of fact, this is another way of reaching consensus through atomic broadcast (*see Securing Total Order* in Section 7.5).

This task can be further complicated if a faulty source can send different values to different replicas (byzantine faults). Note that we have already discussed a means of overcoming this problem. Byzantine Agreement (BA) allows a value to be reliably distributed through a set of consumer replicas under such a failure mode (*see Tolerating Arbitrary Faults* in Section 7.5). If all producer replicas execute BA, then an Interactive Consistency (IC) vector is built (Pease et al., 1980), with exactly the same content at every consumer replica. A deterministic comparison on the IC vector will yield the same result at all replicas.

Exact (bitwise) agreement cannot be reached when two correct replicas can produce different values. How can this happen? Consider, for instance, that the vector of values to be compared is the result of analog sensor readings. Then, any two correct sensors can read values that are slightly different and thus not comparable bitwise. In consequence, one has to use some "convergence function" performed on the whole of the vector, in order to pick the "right" value, which may be neither of the initial values. Besides, in the case of replicated consumers and value faults, the result of the function may be slightly different from replica to replica. This is called *inexact agreement.* Clock synchronization, that we discuss in the Real-Time part of the book, is another example of inexact agreement.

Several convergence functions exist, and some may be sophisticated and highly dependent on the application semantics, for instance, when the value to be produced is a captured image. For the sake of example, we will describe a couple of simple convergence functions for real numbers, that will tolerate up to f faulty values in the vector:

Fault-tolerant Midpoint - selects the midpoint of the values collected after discarding the f highest and f lowest values. Requires at least $2f + 1$ values, $3f + 1$ with byzantine faults

Fault-tolerant Average - selects the average of the values collected after discarding the f highest and f lowest values. Requires at least $2f + 1$ values, $3f + 1$ with byzantine faults

7.8.3 How Many Replicas or Spares?

It is probably worthwhile making a point of the situation in terms of the number of replicas that are really needed to achieve fault tolerance. This number depends on the number and type of faults that need to be tolerated. In order to tolerate f omissive faults, $f + 1$ replicas are needed. To tolerate f value faults, $2f + 1$ replicas are needed, since a majority vote must be made on the value to return. This number is still valid for distributed replicas with value faults, provided that the communication subsystem only does omissive faults. However, it goes up to $3f + 1$, in order to tolerate Byzantine faults.

These numbers represent the minimum number of replicas required. The question now is the following: should we use the bare minimum or should we use additional redundancy?

7.8.4 The Point of Diminishing Returns

Given available resources, one may be tempted to introduce more redundancy than strictly required. This may have several benefits. To start with, it may increase the system reliability, since additional redundancy allows the system to tolerate additional faults. Also, in several replication schemes, additional redundancy may provide load balancing, for instance, by executing reads in parallel on different replicas.

However, additional redundancy also means additional costs and complexity to ensure consistency. For instance, the algorithm proposed in (Lamport et al., 1982) is optimal in the number of rounds ($f + 1$) but requires an exponential number of messages to reach Byzantine agreement ($O(n^f)$). Even load balancing may not be a universal advantage, since it may penalize some classes of operations: in a read-one/write-all strategy to replicated data, the availability of the write operation decreases with the number of replicas.

On the other hand, adding more components means increasing the probability of having a component failed at a given point in time. Thus, after a certain point the introduction of an additional replica may not produce an significant increase in the overall system reliability. It is also important to note that, as we have seen previously, adding additional components is not the only way to increase the system reliability. The other alternative is to use better components. Using analytical models it is possible to find the point where additional redundancy is no longer cost effective and components with higher coverage should be used. This has been called the *point of diminishing returns* (Stiffler, 1978).

7.9 RECOVERY

Most of the techniques that we have discussed so far try to ensure the availability of a correct result. In many systems with less stringent requirements, it is enough to ensure that after a failure the components are able to restart the computation from a consistent state (ideally, not far from the crashing point). Even when availability is a primary concern, it is interesting to ensure that not all of the state is lost in the case where all or some components crash.

Recovery from crashes requires the application state to be saved in *stable storage*. Stable storage entails both the notion of persistence, i.e., surviving the entity creating it, and of reliability, i.e., exhibiting very low probability of losing or corrupting information. Stable storage is usually implemented on non-volatile media (disks, or combinations thereof). However, note that a set of replicated volatile memory repositories can act as a stable storage subsystem. Resilience (persistence and reliability) is secured by ensuring that at least one of those replicas remains operational at all times. The reader should note that passive replication can be seen as a form of "saving state to stable storage".

7.9.1 Stable Storage

Stable storage built from volatile memory is simple and solves several problems, namely as a medium-term repository, for example for long computations, or publisher-subscriber message buses. However, it is not the general approach. This is because truly non-volatile media have a better coverage against unexpected events, such as power breakdowns, and other common-mode failures. How should we build a stable storage device? It is time to apply some of the concepts we have discussed so far, namely the hierarchical nature of fault-tolerant systems, now at the disk level.

The simplest form of stable storage would be a disk. But there are a lot of failures that can affect the information on that disk. Value and space redundancy can be used in order to preserve the consistency of information despite disk errors. For example, the same information can be stored into two different disk blocks and a checksum added at the end of each disk block. If a crash occurs in the middle of a write operation, or if a block is somehow corrupted, the error can be detected from the invalid checksum, and the "last" value can be recovered from the other block. The crash of the whole disk can be tolerated again with space redundancy: the information can be written in two separate disks (this approach is also called "mirrored disks").

An interesting technique that increases both the availability and the performance of the stable storage consists in using a Redundant Array of Inexpensive Disks (RAID) (Patterson et al., 1988). According to this approach, the stable storage container is made of n storage disks and a parity disk. The information is scattered among the storage disks and the parity of the n storage blocks is saved in a correspondent block in the parity disk. If one of the storage disks fails, its content can be recovered by resorting to the parity information. In the current implementations, the parity blocks are also scattered uniformly among all the disks to prevent the parity disk from becoming a bottleneck (this disk would have to participate in all writes).

7.9.2 Checkpointing

The stable store we just studied can be used to save the state of the components that need to survive a crash. The state that is saved in stable store is also called a *checkpoint*. Upon recovery after having crashed, the component reads the last checkpoint and resumes operation from there: this operation is called a *rollback*. As you can see, the basic idea behind this technique, called *checkpoint-based rollback-recovery*, is quite simple, almost too good to be true. However, there are some difficulties that need to be surmounted to render this technique useful.

A very important point that needs to be taken into account is that components do not operate in isolation in a distributed system. They interact with other components by exchanging messages with them. To the recovering process, any messages it sent between the last checkpoint and the crash instant simply do not exist! In some sense, the recovering process will behave like someone who has lost his memory after a car accident and is unable to remember his own wife and children. However, any other component having received those messages will then be inconsistent, since "apparently" no one sent them.

So what is the solution? Fortunately, there is something that can be done with a computer program that cannot be done with humans: travel back in time. Consider the crash and recovery of a component. If all components have periodically checkpointed their state, we can force them to rollback in such a way that some past *consistent global state* is re-established which includes the recovering process. Thus, recovery requires the search for the earliest consistent set of checkpoints, called *recovery line.*

The extensive literature in this area can be classified into three main approaches for taking checkpoints. The first, called *coordinated* checkpointing, consists in having the processes coordinate before taking the checkpoint. The purpose of coordination is to ensure that the set of checkpoints is consistent (*see Consistent Global States* in Chapter 2). The advantage of always taking consistent checkpoints is that, in the case of crashes, the system only needs to rollback to the last checkpoint. On the other hand, coordination itself may introduce some delays in the execution of the application.

Uncoordinated checkpointing avoids coordination costs by allowing processes to take checkpoints independently from each other. Unfortunately, with uncoordinated checkpointing there is no guarantee that a set of consistent checkpoints will exist. We recall that two checkpoints C_1 at process p_1 and C_2 at process p_2 are mutually inconsistent if C_1 contains the message m sent by p_1 to p_2 but C_2 has no record of sending this message. It is easy to see that this scenario may repeat itself: the rollback of one process may force the rollback of another process, which in turn forces the first process to rollback again, and so on. This phenomenon, called *domino effect*, may cause the system to rollback to its initial state (Randell, 1975). The effect is shown in Figure 7.20: p_1 fails, and then recovers, rolling back to checkpoint c_a. Evidence of sending message m_i no longer exists, and so, p_2 is forced to rollback to checkpoint c_b. However, this "unsends" message $m_j$and thus p_3 is forced to pull back to c_c. We leave it to the reader to confirm that *rollback propagation* will bring the system back to the initial state.

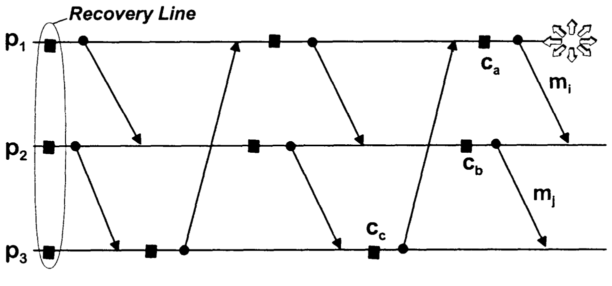


Figure 7.20. Domino Effect

A third technique, also not requiring coordination, is called *communication-induced* checkpointing. It avoids the domino effect by requiring components to checkpoint upon receiving and prior to processing certain messages that may induce conflicts.

Finally, it is worthwhile mentioning that if the checkpoint is made not to a local store, but to one or several additional component replicas in different sites, then it is not necessary to wait for the component to recover after a crash: recovery may start in one of the replicas immediately the crash is detected. This is the mechanism underlying passive replication (*see* Section 7.6).

7.9.3 Logging

Many applications could live with a system that simply checkpoints periodically, often enough that the rollback delay does not become too large. Nevertheless, there are some problems that limit the effectiveness of rollback. The most obvious is when the computation is not fully deterministic: upon recovery the component is not guaranteed to reproduce exactly the same steps and results it has produced before the crash. Also, certain actions done since the last checkpoint may have left a trace outside the subsystem of components involved in the checkpointed computation. Actuations on the environment or messages sent out are good examples of what we may call *real actions*: these cannot be undone, and rolling back and repeating them may cause inconsistent behavior.

This problem could be avoided by systematically checkpointing whenever such an action is performed. However, one of the problems with checkpointing is that it consumes time. If the component holds a large state, or if the above-mentioned actions are too frequent, the performance overhead of checkpointing may become unbearable. Checkpoints may be optimized of course, namely by selecting the state variables to checkpoint, e.g., by using application semantics and/or compressing the data to be stored.

However, if the computation is piecewise deterministic (*see* Section 7.6), there is a systematic way to minimize the number of required checkpoints, by *logging* all non-deterministic events between consecutive checkpoints. The state of the component can then be reconstructed from the most recent checkpoint and the log: the state is first recovered from the checkpoint and then all events from the log are replayed in the same order as before the crash. This technique is called *log-based rollback-recovery.* This may allow the system to recover without forcing other components (or the outside world) to rollback. Three major approaches to log-based rollback-recovery have been proposed: pessimistic logging, optimistic logging, and causal logging.

Pessimistic logging systems make sure that information about each non-deterministic event is logged before the event affects the computation. As a result, when a process crashes and recovers it is guaranteed to execute the same sequence of events, reaching a state that is consistent with the (internal or external) actions performed prior to the crash. Unfortunately, the number of log operations that are inserted in the process path may represent a significant performance overhead. To avoid these costs, it is possible to log information about non-deterministic events asynchronously. This means that the computation proceeds and the logging is performed later. The idea is that since faults are not very frequent, all log operations succeed in most cases. Thus, these protocols are also called optimistic protocols. Unfortunately, once in a while a process may crash before logging all non-deterministic events. Upon a failure, the maximum consistent state must be recovered from the last global consistent checkpoint and from a sub-set of events recorded in the local logs. To achieve this state, other processes may be forced to rollback to obtain a consistent global state. In consequence, interaction with the outside world requires explicit synchronization among processes. The third alternative, causal logging, keeps track of causal relations among events, retaining most of the advantages of optimistic logging without making optimistic assumptions. The description of causal logging is outside the scope of this book (for a survey, see (Elnozahy et al., 1999)).

Finally, garbage collection is also an important matter, since both logs and checkpoints claim resources. Uncoordinated checkpoint protocols identify the recovery line and dispose of all checkpoints past it. Coordinated checkpoint protocols dispose of all checkpoints but the most recent. Logs are deleted immediately a new checkpoint is made.

7.9.4 Atomic Commitment and the Window of Vulnerability

The last sections have discussed recovery approaches based on actions that can be rolled back individually. A step forward would be to encapsulate these actions in sequences that cannot be undone individually, and have the system automatically guarantee this. Atomic transactions achieve this effect: if a transaction commits, its effects are durable. A protocol that ensures such properties is called an *atomic commitment protocol* (*see Distributed Atomic Commitment* in Chapter 2), and what we discuss here is how much we can rely on such a protocol, in the event of process crashes and recoveries.

Recapitulating, the processes participating in a distributed transaction must coordinate their actions (and checkpoints) to ensure that a committed transaction is not erroneously aborted upon recovery. The *two-phase commit* protocol is one of the most used atomic commit protocols (Figure 7.21). We revisit the algorithm with more detail. This protocol is coordinated by one of the participants. In the first phase, the coordinator sends a PREPARE message to all other participants. Upon reception of a PREPARE, a process checks if it is ready to commit the transaction. If the answer is affirmative, it ensures that all the results are logged in stable storage, appends a *prepared* entry to the log and replies OK to the coordinator. Otherwise, it aborts the transaction and replies with NOTOK. If the coordinator receives an OK from all participants it commits the transaction. Otherwise the transaction is aborted. A *committed/aborted* entry is added to the coordinator log and the log is forced to stable storage. The coordinator then initiates the second phase, sending a COMMIT/ABORT message to everybody. Upon reception of the COMMIT/ABORT, each participant commits/aborts the transaction and sends an acknowledgement back to the coordinator. To ensure a fast dissemination of the transaction's outcome, the coordinator retransmits the decision if some acknowledgements are missing.

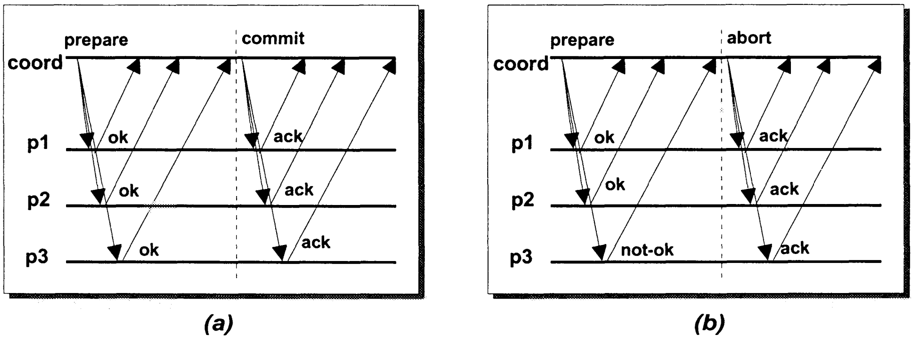


Figure 7.21. Two-phase Atomic Commitment Protocol: (a) commit; (b) abort

The two-phase commit protocol is quite simple and efficient. However, it suffers from a major drawback: if the coordinator fails between the PREPARE and the COMMIT/ABORT, the remaining participants will be blocked waiting for the decision! They cannot abort the transaction either, because the coordinator might have said COMMIT before failing, and a subset of the participants might have committed and then failed. Only when the coordinator recovers can a safe decision be taken. These failure scenarios may also take place if the system partitions.

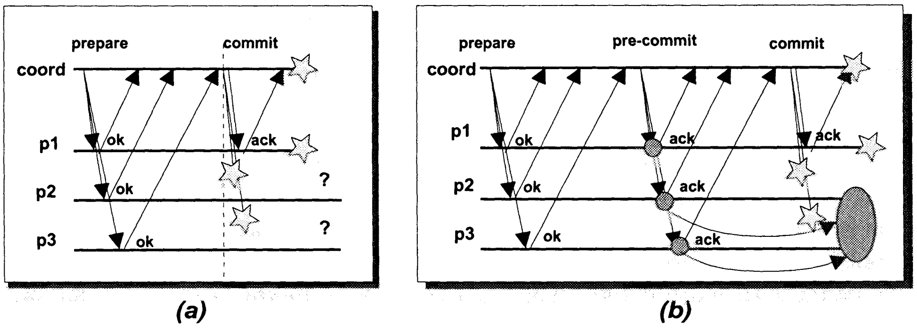


Figure 7.22. (a) Blocking of Two-phase Commit; (b) Three-phase Commit

A non-blocking solution to the problem exists, and is known as *three-phase commit protocol*, exemplified in Figure 7.22b. The idea behind three-phase commit is to delay the final decision until enough processes "know" which decision is about to be taken. Namely, between the two initial phases the coordinator sends a PRE-COMMIT message to all processes and waits for an additional round of acknowledgements. Only then the COMMIT is sent. The advantage of this scheme is that even if the coordinator fails before issuing the commit, the remaining processes may resume the operation since they have received the PRE-COMMIT message. Three-phase commit is much more resilient than two-phase commit, at the cost of performance.

The reader might have already noticed that there are similarities between the atomic commitment problem and the agreement problem. If fact, atomic commitment can be seen as a form of agreement with some restrictions: all participants must agree on the outcome of the transaction, with the proviso that the outcome can only be commit if all participants are ready. Actually, since we want to prevent failed processes from disagreeing with active processes, atomic commitment is a variant of uniform agreement. Non-blocking solutions to the problem of atomic commitment were known before the problem of uniform agreement was well understood. As we have already discussed, there are no deterministic solutions to the (uniform) agreement problem in asynchronous systems. Thus non-blocking actually means: 'non-blocking as long as a majority of processes remain correct'.

Actually, an elegant way of describing an atomic commitment protocol is to use consensus as a building block. The protocol can then be rephrased as follows. The first phase proceeds as before with a minor difference: one participant is responsible for sending a PREPARE message but responses are multicasted to all participants. The subsequent phase is decentralized and based on the execution of consensus. In order to reach a decision about the outcome of the transaction, all participants propose an initial value to consensus. Participants that collect OK from all other participants propose to COMMIT the transaction. Participants receiving a NOTOK or suspecting that some other participant has failed propose to ABORT the transaction. The consensus itself guarantees that all correct participants agree on the same outcome.

7.9.5 State Transfer

We now consider the case when the component recovers and can be re-integrated in the replica group in order to re-establish the original number of replicas. The state of the recovering process has to catch-up with the state of the remaining replicas. This problem requires specialized protocols, called *state-transfer* protocols, whereby one of the active replicas transfers its state to the recovering one. State transfer poses both practical and conceptual problems.

The more practical problems are concerned with the identification, capture and physical transfer of the state. For the sake of performance, one should try to transmit only the variables relevant to the recovering replica. If the replica only has volatile storage, everything must be transferred. However, it may be able to load code and read-only tables from disk, and re-initialize the computation. This minimizes the amount of data to be transferred. Application-specific state-transfer opens further opportunities for optimization, since the programmer knows better than anyone else what to transfer. For example, suppose that the state of a replica depends exclusively on the last n commands it executed: for instance a component that keeps an average of the last k sensor readings. In this case, replica integration can be performed just by waiting (after k sensor readings, the state of the replica is updated).

If replicas checkpoint/log to stable storage, the recovering replica can implement incremental transfer, i.e., just transfer the results of the changes made during the crashed period. This requires the recovering replica to recover first from the last checkpoint/log in stable storage, and then from the active replicas. These have to take the necessary steps to keep a history of changes since the failure of a replica is detected, until it recovers.

The conceptual problem is that ideally state-transfer should be performed with minimal interference with the behavior of the remaining replicas. That is, active replicas should not be prevented from providing service to clients while the state is being transferred. The problem with this approach is that the state is a moving target: it is being changed at the same time it is being transferred! On the one hand, this may entail incorrect system state transfer. On the other hand, if the active replicas change their state faster than they are able to transfer it, the joining replica will never be able to catch-up.

The former problem may be solved by giving priority to state transfer in detriment of replica computations, which may momentarily slow the computation down. As for the latter problem, we now describe a simple way to perform state-transfer in systems where total order is used. The protocol is exemplified in Figure 7.23. The recovering replica (p_3) initiates the procedure by resuming communication with the replica set. At this point, if the set was using some

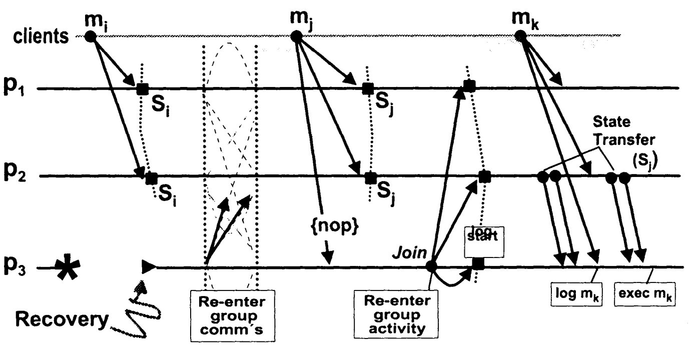


Figure 7.23. Recovery with State Transfer

form of group communication, it starts receiving all messages, but still discards them. Next, it multicasts a request to JOIN the replica group activity, which triggers a state-transfer operation. This message is delivered in total order to all replicas, including the joining replica, marking a cut in the global system state: one of the active replicas (p_2) checkpoints its state at this point (S_j), and sends it to p_3 in one or more STATE messages; p_3 starts logging any messages that arrive after the cut, since they follow S_j. New requests from clients now arrive at all replicas (e.g., m_k), and can continue to be processed by all replicas except the joining one. The joining replica logs all client requests until it receives the last STATE message. Then, it consumes all pending requests in the log, in the order by which they were received. At this point, state transfer is complete, and all replicas are consistent. It should be clearer now why it may be of interest to slow down the computation a bit during state transfer: if the log is enormous after state transfer, the recovering replica may take very long to catch-up. Why is this a problem? Remember that the whole purpose of recovery was to re-establish the degree of replication, which is only achieved when transfer is complete.

7.9.6 Last Process to Fail

Assume a replicated computation following an optimistic replication strategy. As the name implies, availability is the primary concern in this approach, and so as long as one replica is active the service is not interrupted. Replicas have some persistent state that survives failures. As replicas start failing, their persistent state will be made obsolete by the progress of the surviving replicas. If all fail before anyone of them has the chance to recover, the last replica to fail has the most up-to-date state. Upon recovery, operation cannot be resumed before this replica recovers, otherwise the last updates would be lost.

The question is: when a process recovers how can it know that it was the last process to fail? Let us assume that each replica keeps a *version* number that keeps track of the number of changes performed that replica state. If all recovering processes compare their version numbers, the one with the highest version number knows it was the last replica to fail. The problem with this approach is that all replicas need to recover in order to perform the comparison. It would be more interesting to resume the operation as soon as the last replica to fail recovers. How to do this?

A solution exists to this problem (Skeen, 1985). Consider that each process is able to detect the failure of other processes. When process i detects that some other process j has failed, it adds this information to a local "obituary" log which is saved in stable storage. The last process to fail is the process that has registered the death of every other process in its own obituary log.

7.10 SUMMARY AND FURTHER READING

This chapter has discussed the main challenges in achieving correct operation of a distributed system in the presence of faults. Several basic paradigms were addressed, such as: failure detection, partitioning, membership and consensus, agreement and order of message delivery. If these problems are solved, one can tolerate faults through the use of replication. Replication has several coordination aspects, which depend on the underlying system model: partition-free or partitionable; crash or crash-recovery. Resilience and recovery finalized the paradigms addressed in this chapter.

For further study, a fundamental reference on the problem of failure detection in asynchronous systems is (Chandra and Toueg, 1996). The QoS aspects of failure detectors are discussed further in (Veríssimo and Raynal, 2000; Chen et al., 2000). For partitionable programming models, see (Amir et al., 1993a; Cosquer et al., 1996; Babaoğlu et al., 2000). Partitioning in synchronous systems is further discussed in *Real-Time Communication Models*, Chapter 13. More information on consensus can be found in (Fischer et al., 1985; Turek and Shasha, 1992). The problem of group membership has been addressed under several system models, namely (Birman and Joseph, 1987; Cristian, 1988; Kopetz et al., 1989b; Jahanian and MoranJr, 1992), or (Amir et al., 1992; Golding, 1992; Rodrigues et al., 1993).

There is also a huge amount of literature on fault-tolerant communications, in particular, on fault-tolerant group communication, such as (Amir et al., 1993a; Birman and Joseph, 1987; Birman et al., 1991a; Chang and Maxemchuck, 1984; Dolev et al., 1993), or (Kaashoek and Tanenbaum, 1991; Ladin et al., 1992; Moser et al., 1995; Rodrigues et al., 1996; Rodrigues et al., 1998a). Deeper study on fault-tolerant communication paradigms can be found in (Hadzilacos and Toueg, 1994).

About the state-machine approach, the excellent tutorial of Schneider is recommended (Schneider, 1993) and for other examples of active and semi-active replication see (Barrett et al., 1990; Chereque et al., 1992). Examples of the primary-backup approach are given in (Speirs and Barrett, 1989; Budhiraja et al., 1993). Improved tree and grid quorum voting algorithms are discussed in (Kumar and Cheung, 1991). Multidimensional voting techniques have also been studied as alternative representations for quorum sets (Ahamad and Ammar, 1991). For dynamic variants of grid algorithms, see (Paris and Sloope, 1992), which allows the grid to be re-organized based on reachability information. Concerning checkpointing, see (Speirs and Barrett, 1989; Kim and You, 1990; Wang and Fuchs, 1992; Silva and Silva, 1992), or (Elnozahy and Zwaenepoel, 1992a; Alvisi and Marzullo, 1993; Alvisi et al., 1999) for further study. A portable checkpoint protocol implementation tool, based on MPI, is described in (Neves and Fuchs, 1998). An excellent and comprehensive survey can be found in (Elnozahy et al., 1999).

8 MODELS OF DISTRIBUTED FAULT-TOLERANT COMPUTING

This chapter illustrates how the paradigms discussed in the previous chapter can be applied and combined to achieve fault tolerance in an application-oriented way. The chapter starts by introducing classes of fault-tolerant systems that make different assumptions about the system properties, from arbitrary to crash and from asynchronous to synchronous. Then, it discusses strategies for the several approaches to building a fault-tolerant architecture. The main models for building fault-tolerant systems are then presented: remote operations, event services and transactions.

8.1 CLASSES OF FAILURE SEMANTICS

We have discussed in Chapter 6 that failures can be classified according to many different criteria. We have seen in Chapter 7 that the solution for a given problem, for instance distributed agreement, strongly depends on the class of failures the system is subjected to. In this chapter we consider the four main broad classes of failure semantics (or failure modes): arbitrary, crash, omissive, and crash-recovery. More refined classes may be derived from the former to satisfy particular assumption requirements.

8.1.1 Arbitrary Failures or No Assumptions

If you build a system under the assumption that components can fail in an arbitrary manner, you are walking on the safe side of the road. Of course, you still have to forecast how many failures can occur, in order to determine how much redundancy to use. Likewise, some environmental and/or infrastructural assumptions are still likely to be made, such as drift of clocks, for example. Nevertheless, the arbitrary assumption approach strongly reduces the chances of unexpected failures, because the coverage of the general system tends to one. Unfortunately, a safer road is also a longer road. Just recall the high number of nodes, messages and rounds required to solve Byzantine agreement (*see Tolerating Arbitrary Faults* in Chapter 7).

On the other hand, assuming arbitrary failures leads to architectures with the lowest performance/price ratio. *Then, when should we choose this model?* To start with, when failures can be catastrophic, and this usually means when human lives or huge amounts of money may be at stake. The highest possible coverage is desired for these systems. Secondly, when the system will be under the threat of malicious attacks. The intruder may manipulate some components in an unpredictable way, attempting to defeat the systems defenses. The arbitrary failure mode implies that rather than trying to guess the intruder's moves, the system is prepared to tolerate any type of behavior.

8.1.2 Fail-Silence or Crash

On the other extreme of the spectrum, the fail-silence mode, often called crash mode, assumes that a component works perfectly until the moment when it suddenly dies, or *crashes*. It does nothing, good or bad, after crashing. This is a widely used failure semantics, specially applicable to components such as processes or processors, which tend to fail in this way. Note that this does not mean that erroneous events cannot happen internally to the component before it crashes. It only means that we expect the internal erroneous behavior associated with the fault never to become visible at the component interface(s). For instance, the address space of a process may be corrupted in an arbitrary manner just before a crash due to an address violation exception; it still looks like a crash if the process does not produce any erroneous output.

The fail-silence assumption simplifies the design of fault-tolerant protocols (*see Fault-Tolerant Communication* in Chapter 7), but should be used with care. Many designers are tempted to postulate that off-the-shelf components are fail-silent, just because they do look as if they were (most of the times). However, experiments have shown that the coverage of this assumption is not very high. In consequence, the system architect should not expect too much from the dependability of systems built to this assumption, unless specific fault-prevention measures are embedded in the design of the system components (e.g., self-checking, wrapping). The level of sophistication of these measures depends of the desired coverage: from hardware (e.g., parity checks, watchdogs) to programming (e.g., systematic validation of types, bounds, message fields, etc.).

8.1.3 Weak Fail-Silence or Omissive

The fail-silence assumption is a bit too strong since it does not consider frequent failures in the omissive class, such as omissions or timing, which are extremely common for certain components in distributed systems, and certainly unavoidable in simplex (non-replicated) networks. The *weak fail-silence* mode establishes the following behavior for a component: (a) it is correct, or else fails by crashing; (b) 'correct' is a non-ambiguous specification of controlled omissive failures. For instance, a weak fail-silent network component specification could be: "during a reference interval, it does at most k omission failures or else it crashes". This could be read as: "components fail by crashing, but while alive they may give up to k successive omissions", which happens to be very appropriate to describe the behavior of most networking components of a distributed system.

Techniques to tolerate omissive faults are easy to implement, and in consequence, it is little more complex to design weak fail-silent systems than it is to design their pure fail-silent counterparts. However, the coverage of properly chosen weak fail-silence assumptions is bound to be higher, for the same system complexity. Many practical systems assume a mixture of fail-silent and weak fail-silent components. Generalizing to less benign failures, whenever a component exhibits a definable failure mode that is lesser than arbitrary, we say it is a *fail-controlled* component.

8.1.4 Crash-Recovery

Most systems assume that components recover sooner or later. With some notable exceptions (such as probes sent to remote locations of the solar system), one can repair or replace failed components. Even in spatial vehicles, some components can reset and recover, such as in the Mars Rover story (*see Resource Conflicts and Priority Inversion* in Chapter 12). The difference between the crash and the crash-recovery modes is that in the first model, the recovering component has no memory of its past existence and has to start anew, whereas in the crash-recovery model the recovering process preserves some of its old state (e.g., up to the last checkpoint made).

Algorithms for the crash (no-recovery) model are concerned with keeping the consistency of processes that remain alive and rely on state-transfer mechanisms to re-integrate recovering components. Algorithms for the crash-recovery model extend consistency requirements to failed processes. Thus, the later tend to require the use of the "uniform" versions of distributed algorithms (e.g., uniform atomic broadcast or uniform consensus, *see Consistency* in Chapter 2).

8.1.5 Synchronous and Asynchronous Models

Timing assumptions are extremely important in a distributed system because they are directly related with the problem of accurate failure detection. With this regard, distributed systems have been classified into synchronous or asynchronous systems, depending on whether or not known bounds on processing, communication delays and clock rate of drift exist (*see* Sections 3.3 and 3.4 in Chapter 3).

The main issue concerning the choice of model with regard to fault tolerance is coverage. Designers frequently postulate a synchrony model (e.g., fully synchronous) and then go about making their design, focusing on the failure assumptions and algorithms to handle them. Very often, the design fails not because some severe fault occurred or too many faults took place, but simply because the synchrony assumptions of the model were violated and that caused the algorithm to misbehave (e.g., misuse of timeouts in the fully asynchronous model, which is in essence time-free; or neglecting timing failures, in the fully synchronous model). Partial synchrony models are in between the former two, trying to take the best of both worlds (*see Between Synchronism and Asynchronism* in Chapter 13). They seem to have an instrument that neither of the others have: they assume *timing failures* in the model. This allows to integrate the synchrony and failure models, making possible a seamless design for fault tolerance.

8.2 BASIC FAULT TOLERANCE FRAMEWORKS

Fault tolerance frameworks provide the grounds for the architect to start the construction of a fault-tolerant system. This section discusses the several vectors along which the architectural work on distributed system fault tolerance may develop. Namely: hardware FT; software-based hardware FT; software FT; communication. In the course of the discussion, it will naturally establish pointers between the paradigms discussed in the previous chapter, and the models to be discussed further ahead.

8.2.1 Hardware Fault Tolerance

The aim of hardware fault tolerance is to tolerate hardware faults using hardware mechanisms. It usually consists of having low-level mechanisms to detect and recover or mask errors. The simplest one is self-checking, which we introduced earlier in this part in Figure 6.7a. Examples of self-checking mechanisms that can be implemented in an efficient way by hardware are parity bits, verification of assertions, checking ranges of memory addresses, etc. When a component is detected to be failed it is stopped. In result, either the system is halted or the system has enough redundancy to continue operating. One approach is to keep a set of spares aside, and enhance the system with a new component, the *supervisor*, which performs a *switchover* from the failed component to a spare, in an automatic manner. When the spare is only started after the failure of the original component is detected, this is called "cold standby", "hot standby" if otherwise. When the technique is applied to stateless components, the cold standby can provide service as soon as it is started. For stateful components, the standby must first "catch-up" with the state of the failed component at the moment of the failure, the *takeover* process, which may entail a glitch in the provision of service.

A natural evolution of the previous approach consists in having two or more component replicas operating in parallel. For instance, if three replicas are used, and their results compared, the unit that compares the results can mask a single error without any service interruption (when the results differ, the majority is assumed to be right). This type of architecture, which we have illustrated earlier in Figure 6.7b, is called Triple-Modular-Redundancy, or simply TMR, and in this context the comparison unit is called a voter. Voting is usually performed at the bit level, which also means that the replicated components operate in lock-step. The TMR architecture can be trivially extended to tolerate more faults by increasing the number of replicas (N-Modular Redundancy, or simply NMR).

In a distributed systems context, hardware fault tolerance today should rather be seen as a means to construct *fail-controlled* components, in other words, components that are prevented from producing certain classes of failures, and then to use these improved components to achieve more efficient fault-tolerant systems.

8.2.2 Software-Based Hardware Fault Tolerance

Software-based fault tolerance aims at tolerating hardware faults using software techniques. Recall that it is the basis of modular FT, which underpins the main paradigms of distributed fault tolerance. In consequence, it is not surprising that all paradigms discussed in the previous chapter are pertinent to this framework. The fault-tolerant computing models that we are going to address in this chapter do an intensive use of software-based fault tolerance. The main players are software modules, whose number and location in several sites of the system depends not from construction constraints, but from the dependability goals to be achieved.

As we studied in Chapter 7 (*see* Sections 7.6 and 7.7), a great deal of the redundancy management policies for software-based fault tolerance inherit basic concepts of hardware fault tolerance, duly generalized, expanded and adapted. Detecting that a component failed and stopping its operation can also be done remotely in distributed systems (*see* Section 7.1). One possible way to detect a failure is to test the component periodically: this may not prevent the failed component from producing erroneous results, but it limits the duration of the abnormal behavior.

Another solution is to force all component outputs through a filter able to assess the correctness of the produced results: if an incorrect output is detected the component is halted. Spare components and replicas can be adequately placed in the system to recover from errors. Secondary spares may be called into operation when the primary fails. Replicas may operate in parallel, in what is called active replication. Besides providing glitch-free operation in terms of availability with regard to crash failures, if replicas diverge this means that an error occurred. A dual configuration may work as a self-checking (albeit distributed) component, whereas a triple or more can provide continuity of service with regard to value failures. When components exhibit fail-silent behavior and there is a need for sparing resources, computation may be based on a primary replica, whereas the backups will be in a warm standby state, lagging the state of the replica. This is passive replication, and the lag is bounded by having the passive replicas be updated from time to time with the state of the primary.

From what was said in the last section, it is evident that software-based and hardware-based fault tolerance are not incompatible design frameworks. As studied in the previous chapter, distributed algorithms that tolerate arbitrary faults are expensive in both resources and time. For efficiency reasons, the use of hardware components with enforced controlled failure modes is often advisable, as a means for providing an infrastructure where protocols resilient to more benign failures can be used.

8.2.3 Software Fault Tolerance

Software fault tolerance aims at tolerating software faults. It is worth to distinguish three types of software faults:

- Faults due to particular sequences of events that are difficult to reproduce (the Heisenbugs). This type of faults can be tolerated by executing the same code more than once in the same or in different machines.
- Those design faults that may lead all replicas to fail in exactly the same way. These faults can only be tolerated by using design diversity. Naturally, the redundancy can be applied in the time or space dimensions, i.e., different versions of the same program can be executed sequentially on the same machine or in parallel on different machines.
- Faults that are specific to some peculiar hardware or configuration parameter (e.g., O.S.). This type of faults can be tolerated by executing the same program on different machines/environments. This case is not very common since it only has advantages if the software just does environment-specific errors.

Extensive bibliography has been published on the subject of software fault tolerance since the pioneering works described in (Randell, 1975; Chen and Avizienis, 1978). See for example in (Kim and You, 1990; Issarny, 1993; Xu et al., 1995), or the survey in (Lyu, 1995).

8.2.4 Fault-Tolerant Communication

Communication is so specific of distributed systems that it prefigures a framework of its own. Several techniques assist the design of fault-tolerant communication networks, as we saw in Section 7.5. Their choice depends on the answer to the following question: *What are the classes of failures of communication network components?* This is akin to establishing the failure mode assumptions, and leads to the selection of the type of redundancy: space, time, or value.

Assertive errors can be detected using value redundancy, in the form of message consistency checks, such as CRCs or signatures. Corrupted messages can then be dropped, transforming an assertive fault into an omission fault. If enough redundancy is provided, the errors can be not only detected but also corrected: the combination with time redundancy, by repetition, recovers or masks most errors. Some semantic errors require space redundancy in order to be masked, namely because they can be made before the insertion of the consistency check.

Omissive errors are extremely common and they form the bulk of the body of research in FT communication. Timing errors are mainly addressed through three schools of thought: the one that seeks at preventing their visibility at system level, normally related with hard real-time systems design; the one that attempts at recovering from or masking them, sometimes with application help, akin to mission-critical systems design; and the one that gets rid of them, by assuming that the system is time-free, related with asynchronous systems design (*see Real-Time Communication* in Chapter 12 for the former two). Omission errors are classically addressed by time redundancy. Space redundancy by full replication is only used when either the glitch associated to recovery or the bandwidth overhead generated are not desired.

Finally, some network components can crash permanently. This may occur at the interface between the node and the medium (for instance, a malfunctioning Ethernet card) or at the medium itself (a broken cable, a broken repeater, etc). The effect of a component crash on the overall network heavily depends on the topology of the network, technology used, medium access protocol, location of the error, etc. It may disconnect a node, a partition, or the complete network. As for omissions, the chosen policy depends on the desired quality of service: full redundancy by duplication or n-plication of all network components; or medium-only redundancy, either by reconfiguring or replicated medium components.

8.3 FAULT TOLERANCE STRATEGIES

There are several factors affecting the choice of strategy to design a fault-tolerant system, such as: classes of failures (i.e., aggressiveness of environment); cost of failure (i.e., limits to the assumed risk); performance/price ratio; available technology (i.e., limits on the FT constructs available). We line up below the strategies we feel as most important. Once a strategy defined, design should progress along the guidelines suggested by the several fault-tolerant design frameworks just presented. Strategic issues are: redundancy policies between fault prevention and fault tolerance; types of faults tolerated (i.e., hardware or software faults); level of service provided (i.e., glitch-free or recoverable operation). We are going to discuss the main strategic issues concerned with fault tolerance in distributed systems: fault tolerance vs. fault avoidance; tolerance of design faults; perfect non-stop operation; reconfigurable operation; recoverable (fail-fast) operation; fail safe vs. fail operational.

8.3.1 Fault Tolerance versus Fault Avoidance

Fault avoidance, in its facets of fault prevention and fault removal, is an important part of resilient system design, aiming at amplifying component reliability. Given the complementary nature of fault tolerance (FT) and fault avoidance (FA), how much emphasis should be put on each in relative terms? Is it better to use highly reliable (but also highly expensive) components? Or should we use several copies of less reliable components and pay the overhead of complex fault-tolerant protocols?

The good mix of FT and FA has to do with two issues: the number, and the severity of faults. In terms of the number, the answer lies basically on the expected length of the mission versus the expected reliability of the components used, for a given amount of redundancy. For a given reliability, the number of spares should not go beyond the point of diminishing returns, and this dictates the balance. The issue is also relevant when space, power-consumption or performance reasons discourage the use of large amounts of replication.

However, the touchstone of the FT/FA balance lies in the nature of faults, vis-a-vis the efficiency of the associated replica consistency protocols. We have seen in Chapter 7 how costly protocols tolerating arbitrary faults can be, with regard to protocols tolerating omissive or even certain kinds of assertive faults. Based on the quality of service to be provided by the system, and on a preliminary elaboration of failure mode assumptions about the environment and infrastructure, the architect should lay down her strategy, starting with the definition of the architecture and the choice of components. Then, the evaluation of the coverage of assumptions may dictate successive iterations through this process until a balance between coverage and effectiveness is obtained. A cost-effective approach has been to rely on Commercial Off-The-Shelf (COTS) components. Very often, COTS do not offer the desired level of confidence in their behavior. In such a case, fault tolerance may recursively be applied at sub-system level, enhancing those COTS components in order that a controlled failure mode is obtained of the final ensemble, with high enough coverage. This recursive use of fault tolerance to build the component effectively classifies as fault prevention at the interface of the component, when seen at the outer system level.

8.3.2 Tolerating Design Faults

Most of the design faults in mature systems, software or hardware, are transient faults. As such, many fault-tolerant systems that use replication or time redundancy (repetition or re-execution) become tolerant of design faults, since subtle design faults tend to be activated only in some scenarios which are hard to reproduce. This is a valid strategy for all but highly-critical systems. In this case, one has to rely on design diversity which is naturally extremely expensive. In the limit, the architect will point to the development of N code versions, each produced by a different team, which will execute in as many distinct hardware platforms, in active replication. Note that it is usually harder to build a system using diverse components, because of problems such as conversion between different machine representations of equivalent values.

8.3.3 Perfect Non-stop Operation?

Is there such thing as perfect non-stop operation? This notion is in the eye of the beholder, that is, the service user, and so a careful answer will be "in principle, yes".

The ideal fault-tolerant system masks all errors in such a way that the user is never aware of their effect. Note that the masking approach is expensive, and is eligible when the cost of fault tolerance is negligible compared to the cost of service interruption. It is thus the strategy for life- or money-critical systems, such as flight control or air traffic control, on the safety and availability facets, respectively. Fault tolerance is something like virtual memory: it requires better hardware and slows down the execution a bit, but it is hard to live without it.

Network omissions and some software Heisenbugs can be masked using time redundancy. The most straightforward approach to mask crash failures is to use active replication, for example under the diffusion or event-based model, which we address in Section 8.5. It should be noted that in some cases even backward recovery is adequate, if the recovery glitch is short enough to go unnoticed. Sometimes when we use the Web, our network connection is so slow that we wouldn't notice the server crashing and recovering.

In partitionable or pure asynchronous systems, perfect non-stop operation is not achievable unless in some very specific cases where the application semantics imposes little consistency requirements (recall the FLP impossibility result!). Thus, if non-stop operation is required it is necessary to invest on redundant network architectures (that minimize the probability of partition) and time-controlled environments (to ensure the required synchrony).

8.3.4 Reconfigurable Operation

Active replication is expensive and as such many services resort to cheaper redundancy management schemes, based on error recovery instead of error masking. This alternative approach can be characterized by the existence of a visible glitch. The underlying strategy, which we may call reconfigurable operation, is normally addressed at availability-oriented services, such as transactional databases, web servers, etc., normally accessed through remote operation primitives, such as studied in Section 8.4.

Several techniques may be used, and the typical replication management schemes fall in the semi-active and passive classes (*see* Section 7.6). These techniques imply an omissive failure mode, and are normally used to handle crash failures of components. The failure of a component triggers a reconfiguration procedure that automatically replaces the failed component by a correct component. During reconfiguration the service may be temporarily unavailable or suffer some performance degradation, whose duration depends on the policy used. For example, the take-over of a passive backup when the primary fails produces a glitch composed of the detection and the reconfiguration latencies. Reconfigurable operation is very complex in asynchronous systems, because it is impossible to have accurate failure detection and one risks to have multiple components playing the role of "primary", if the adequate algorithms are not used (*see* Section 7.7).

8.3.5 Recoverable Operation

Consider that a component sometimes crashes but recovers after some time. A fault-tolerant design can be obtained under these circumstances, if a set of pre-conditions hold. Firstly, the duration of recovery must be known and bounded, and short enough for the application's needs. Secondly, the crash must not give rise to incorrect computations. This may achieved through several techniques, amongst which we name checkpointing into stable storage and logging executions past the last checkpoint, as studied in Section 7.9. In distributed computations, NVRAM may provide the support for logging last moment state and achieving consistency of recoverable remote operations (*see* Section 8.4). Recoverable exactly-once operation can be achieved with atomic transactions, which we address in Section 8.6.

This strategy concerns applications where at the cost of a noticeable temporary service outage, the least amount of redundancy is used. Architectures have evolved in order to grant a reduced recovery latency, in what are also called *fail-fast* systems. The strategy also serves long-running applications, such as scientific computations, where availability or reliability concerns are not as demanding as in interactive applications. However, the duration of the computation is such that the probability of a failure jeopardizing the whole computation requires attention.

8.3.6 Fail-Safe or Fail-Operational

In certain situations, it is necessary to provide for an emergency action to be performed in case the redundancy within the system is no longer enough (spare exhaustion) or is not adequate for the faults occurring (assumption coverage). In this case, rather than letting the system evolve to a potentially incorrect situation, which in critical systems may entail a catastrophic failure, it is preferable to shut the system down at once, what is called a *fail-safe* behavior.

This strategy, important to safety-critical systems, may complement other strategies concerning the regular behavior of the system. As a variation, instead of halting at once certain applications may require that the system performs an orderly shut down routine before halting. In this case, it is required that no matter the situation that causes the need for shut down, the system have the capability of performing the orderly shut down routine, as part of its nature of being a fail-safe system.

In special cases, the fail-safe nature of a system is not obtained by stopping it at all. Indeed, in some cases, that would be a catastrophic outcome, as in the case of a flying airplane with engine problems. In these situations, there must be contingency plans for the system to be put in a mode where it continues operating despite having failed the provision of its intended service. This should be achieved at least until a safe stop is possible. We call these systems *fail-operational.*

8.4 FAULT-TOLERANT REMOTE OPERATIONS

Remote Procedure Calls (RPCs) or Remote Method Invocations (RMIs) are a well established method of providing a reasonable degree of distribution transparency to the application programmer. Basically, RPCs are based on a client-server architecture. Communication is supported by a simple message exchange protocol that operates in a request-reply fashion. In this section we discuss the fault-tolerant issues related with the implementation of fault-tolerant RPC services.

8.4.1 Reliability of Remote Operations

An RPC architecture has mainly three macroscopic components, namely: the client, the network and the server. Of course, each of these components can fail. To illustrate the main issues regarding fault tolerance in RPC systems we first assume a relatively benign (and common) failure mode: clients and servers can fail by crashing and the network can crash and/or lose messages.

What is desired of the server is that it executes requests once, and only once. This desirable semantics is called *exactly-once*, but we will see that it is unreachable in the case of remote operations with RPC in the presence of failures, and we will have to accept weaker semantics. The several failure scenarios and possible remedies are illustrated in Figure 8.1. We will refer to them in the discussion below.

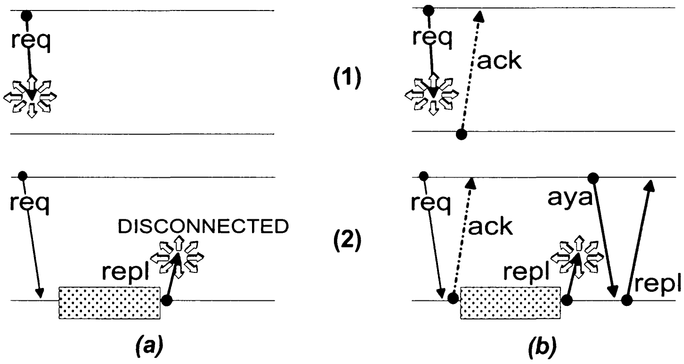


Figure 8.1. Remote Operations: (a) Network Failures; (b) Remedies

Network omissions may affect either the request sent from the client to the server or the reply sent back by the server to the client (Figures 8.1.1a and 8.1.2a). These two faults are indistinguishable from the client point of view; in any case, what it can observe is the absence of the desired reply. Well, this protocol was too simple, and thus we consider enhancing it, in the way shown in Figures 8.1.1b and 8.1.2b: the request is acknowledged, and this remedies the first problem, since the client can timeout on the acknowledgment and repeat the request; for the second problem, the client periodically enquires the server with an "are you alive?" heartbeat message, which the server replies with the RPC result if it had already finished.

So far so good, but the server may fail. See Figures 8.1.1c and 8.1.2c, depicting two distinct situations: the server fails *before* executing the service in the former, and *after* executing it in the latter. Although the enhanced protocol detects server failure through the *aya* heartbeat, the two situations are indistinguishable from the client viewpoint. Worse, it may even happen that the request arrives, it is executed, the *ack* is lost and then the server fails: the client still thinks that the request did not make it to the server. In short, the client may identify similar syndromes for different failures. Is this a problem?

Intuitively, if the client does not have a result, it makes sense for it to re-issue the request until a reply is received or until some pre-defined number of retries has been exceeded. However, this uncertainty leaves the client in suspense: "Did my Pizza order succeed or should I buy a Burger?". The problem of having the client re-issue the request is that the server may receive several copies of it. In the case of the Pizza order, the client may not be willing to pay for four Pizzas when she did order just one (no pizza is *that* good), so discarding the duplicate requests to ensure only one execution seems like a wise decision.

However, enforcing this behavior introduces a non-negligible amount of overhead. Requests must be stamped with some unique identifier and the server must store the identification of previous requests. Additionally, the server must also store the replies it has produced in the past, in order to send them back to the clients when requested. Even if clients make their requests one at a time (which means that the server just has to keep the reply to the last request from each client) this may still be a significant overhead when a server has many clients.

An additional problem will definitely drive us away from exactly-once semantics: all that was said above works while the server does not fail. Consider a simple server, with no stable storage: it is amnesic after it recovers, and thus it cannot know which requests it has executed (recall that the client does not know either). There are only two alternatives: either execute request repetitions, or do nothing.

If by all means re-execution must be avoided, then a recovering server cannot execute request repetitions. This discipline combined with duplicate elimination happens to be the strongest semantics that can be achieved with RPC remote operations, called *at-most-once*: it does what its name implies, and thus, maybe nothing ends-up being executed, or worse, something is partially executed. Some researchers further distinguish the first situation, calling it *zero-or-once* behavior.

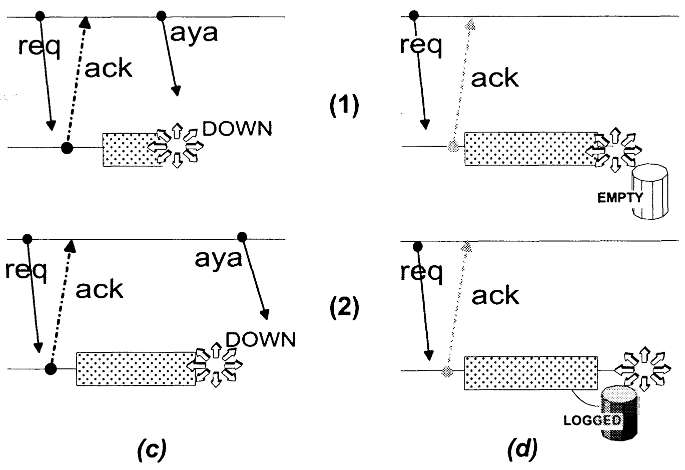


Figure 8.1 *(continued)*
. Remote Operations: (a) Server Failures; (b) Logging to Non-Volatile Store

The effectiveness of at-most-once semantics can be enhanced by making servers state-full. If they preserve their state in persistent memory, they can reconstruct it upon recovery before resuming service. This has severe performance impact, and thus a practical implementation consists in using NVRAM[1] rather than disk, just to keep request identifiers and their execution state. This reduces the window of ambiguity concerning request execution state.

Yes, we said reduce, not cancel. See Figures 8.1.1d and 8.1.2d, depicting two very distinct situations taking place after the server executes the request: it fails *before* logging it in the former, and *after* logging it but before informing the client, in the latter. We leave it to the reader to understand that there is no way out of this ambiguity, except by using another paradigm.

State-full servers and at-most-once behavior not only consume more resources but may also exhibit poor performance. This overhead should be avoided whenever there is no significant disadvantage in executing the same request more than once. For instance, consider a server that provides clients with the value of the room temperature. Executing the same request several times has no negative effect other than consuming resources on the server machine.

Requests that can be executed several times and produce equivalent results are said to be *idempotent*, and the corresponding semantics is called *at-least-once*. This is so convenient that is often worthwhile designing the application in such a way that all requests have this property. In this case, servers are

[1]Non-Volatile RAM.

much simpler and faster, since they are not required to keep any state about past requests. In consequence, they have no state to lose when they crash. For this reason, they are said to be *stateless*. Stateless servers can recover after a crash and continue providing service as if nothing happened.

Client failure is less critical from the point of view of data consistency, but still deserves some comments. If a client fails, this is only a problem if the server has given the client the exclusive right to use some resources, in the course of execution of a previous request. In a fault-tolerant system, to assign resources to a component that may fail and whose failure cannot be reliably detected is generally a bad idea. A more conservative approach is to *lease* (Gray and Cheriton, 1989) the resources for some fixed amount of time and require to client to re-acquire the resources periodically. Another downside of having clients whose crash goes unnoticed is that their last wishes may keep the servers busy computing responses that nobody will ever read. These computations are called *orphans*, and the problem may be solved with a facility called orphan detection (Panzieri and Shrivastava, 1988).

8.4.2 Building Reliable Client-Server Systems

So far we have discussed the issue of data consistency and operation correctness in Remote Procedure Call systems, in the presence of failures. We often want more than that: we also want availability. If modular fault tolerance is used, the issues concerned with building a reliable client-server system are incremental to the principles of building basic C-S systems discussed in *Client-Server with RPC*, Chapter 3.

It is time to apply the replication techniques that we have studied in previous chapters. A very intuitive way of offering high availability in an RPC system consists in constructing a fault-tolerant server using the replicated state-machine approach. The client broadcasts the reply to the group of servers using a primitive that enforces agreement and order. An atomic multicast protocol is particularly well suited for the job. All replicas process the request and send back a reply to client. According to the types of faults being tolerated, the client can pick the first reply or wait for a majority of similar results. The servers can be themselves clients of other servers. Since the server is replicated, different copies of the same request will be produced which need to be combined in a single correct request (again, using voting) before being processed by the servers.

The actively replicated state-machine approach is the most intuitive way of building reliable client-server systems using replicated RPCs. However, it is not the only one. All the other schemes discussed before, namely the semi-active and passive replication schemes can be used (and have been used) to build fault-tolerant C-S servers.

8.5 FAULT-TOLERANT EVENT SERVICES

Sometimes, it is more convenient to express an application in terms of event notifications than through request-reply interactions. As we have seen in Part I of this book, event-based systems, like their RPC based counterparts, have three main components: the event producers, or *publishers*, the communication media, or *channel*, and the event consumer or *subscriber*. Two variants of the architecture can be found: volatile channel systems (where the messages can be consumed as they cross the channel and discarded afterwards) and persistent channel systems (where the messages are kept by the system until consumed by their recipients). Fault-tolerant techniques can be applied to both systems.

8.5.1 Volatile Channel Architectures

In event based systems, when a notification is produced it is usually with the aim of being processed by some other component interested in the event. The processing associated with the event may be rather simple, such as storing a sensor reading in a log, or complex, such as shutting down the system in a graceful manner after an alarm has been fired. In such systems, fault tolerance means: (i) ensuring that the desired event is produced; (ii) that it is reliably delivered to the consumer; and (iii) ensuring that there is a consumer ready to process the event.

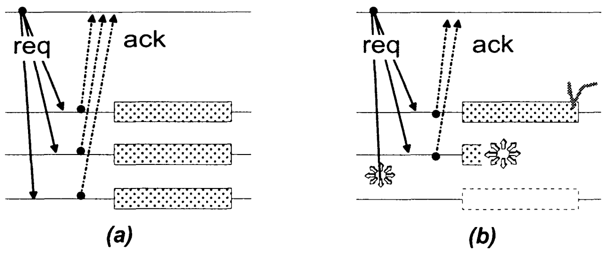


Figure 8.2. Replicated Channel: (a) No Failures; (b) Exactly-once Execution

In order to ensure that the event is produced, replication should be used. For instance, if a sensor reading must be available, one has to replicate the sensor. In order to ensure that the event is processed, one can also replicate the subscribers. As before, to keep the replicas consistent, the replicated-state machine approach can be used. Naturally, this requires the use of group communication in order to ensure the all events are delivered in total order to all consumers. In terms of failure semantics, note that it is quite straightforward to achieve *exactly-once* behavior with the state machine model fed by reliable group communication. Observe Figure 8.2: in the no-failure situation we have all three replicas execute the request; three replicas survive 2 failures and deliver the service exactly-once, as shown.

8.5.2 Persistent Channel Architectures

Persistent channel publisher-subscriber architectures are similar in many respects to volatile channel architectures. However, since the channel stores the events for later retrieval, the architecture does not require the publishers and the subscribers to be active at the same time in order to exchange notifications. On the other hand, the channel is much more than just a communication media, it is a storage media that needs to preserve the messages in a reliable way.

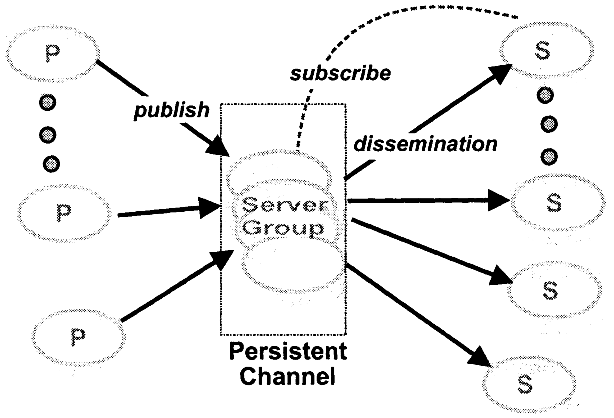


Figure 8.3. Fault-Tolerant Publisher-Subscriber

One way to describe the persistent channel abstraction is to consider the channel as a fault-tolerant server that can be accessed by publishers and subscribers. In some sense, the "channel server" implements some form of stable storage that is fault-tolerant, namely to crashes. The channel can be implemented as a replicated set of servers, using techniques described earlier. This can be done either through non-volatile storage, or through replicated volatile storage (*see Recovery* in this Chapter). Figure 8.3 exemplifies the principle: the message channel is at least capable of recoverable operation without losing data. Since it is replicated, it can also achieve non-stop operation given a sufficient degree of replication. Publishers and subscribers can access the server using one or several IPC primitives (message passing, RPC, group communication).

8.6 TRANSACTIONS

This section describes a number of relevant issues regarding the implementation of transactional systems. The concept of concurrent transaction has been introduced in Part I of the book (*see Concurrency* and *Atomicity* in Chapter 2). We begin by summarizing the most important notions, before we delve into the implementation decisions concerning transactional systems.

Most applications have sequences of operations that are only useful if executed as a whole. Consider for example a bank transfer that withdraws money from one account and deposits the same amount in another account. If this sequence is interrupted by some reason (for instance, a failure) someone will lose money (either the money is withdrawn and never deposited or vice-versa). At the application level, recovery can be greatly simplified if some underlying mechanism guarantees the *atomicity of failure* of a sequence of operations. Atomic transactions are such *indivisible* sequences of operations: either all operations in the sequence are successfully executed, or none is executed. We say that a transaction *commits* its results when it successfully terminates; otherwise we say that the transaction *aborts*. Transactional systems must have some way to delay the effects of the transaction until the transaction commits (i.e., they keep the intermediate results in a log, also called the *redo log*). Alternatively, operations may be allowed to take effect immediately, as long as the system has a way to *undo* these effects, in the case the transaction aborts (what is called an *undo log*). Note that if another transaction reads a value produced by a transaction that is later aborted, what is called a *dirty read*, the second transaction must abort too (it has "seen" results that, for all practical effects, "did not happen"!).

Clearly, transactions are easier to implement in systems where the semantics of the operations are simple and well understood. For instance, if all operations are *reads* and *writes* in data items, it is almost straightforward to construct a redo log (containing the new data values) and/or an undo log (containing the old data values). In fact, databases are the ideal field of application of transactions. In systems that perform external actions, also called real actions, enforcing transactional semantics is much more complex and sometimes impossible.

The atomicity and indivisibility of the transactions also have another facet: concurrency in the access to the same data by different transactions should be "hidden" from the transaction programmer. This preserves the intuitive notion that the transaction is indivisible and executes as a single atomic instruction. Also, it is well known that concurrent programming is by no means a trivial task, which is handled automatically by the transactional system. Getting *concurrency control* out of the way of the application programmer is by itself an effective way of improving code reliability and guaranteeing data *consistency*. Finally, the effects of the transactions should not be lost, or in other words, must be *durable*. Of course, how much durable depends on the applications needs. There is a range of progressively more fault-tolerant approaches: the results are stored on disk; on redundant disk arrays; or in several different disks on different locations.

Collectively, the "**A**tomicity, **C**onsistency, **I**solation, and **D**urability" properties are also known as the ACID properties of transactions.

8.6.1 Transaction System Architecture

Transaction systems are generally composed of three components, namely: the *Transaction Manager*, the *Scheduler* and the *Data Manager*. A generic block diagram of a transaction system architecture is depicted in Figure 8.4, showing the interaction between the several modules.

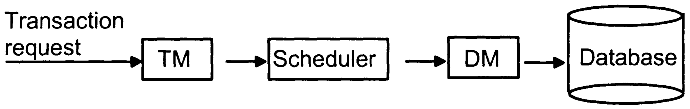


Figure 8.4. Transaction System Architecture

```
procedure Deposit (inout Account a, in Amount v)
begin
    Amount x;

    x = a.read ();
    x = x + v;
    a.write (x);
end Add;
```

Figure 8.5. Deposit transaction

The Transaction Manager ensures the interface with the application code, adding transaction identifiers to the application requests and forwarding them to the appropriate node. The Scheduler is responsible for concurrency control; it executes, rejects or delays the operations to enforce concurrency control. The Data Manager is itself composed of two sub-components: the *Recovery Manager* and the *Cache Manager.* The Recovery Manager is responsible for the resilience of the data accessed by the transaction; it manages the physical media, the recovery logs and applies the recovery procedures. The Cache Manager is responsible for moving data between the stable storage and the faster volatile memory.

8.6.2 Concurrent Transactions

Those that have experience with concurrent programming are quite familiar with the problems that can occur when several threads of control concurrently access a shared data structure. Transactions are no exception and also require the use of some form of *concurrency control.*

Let us give a couple of simple examples of problems that may occur if concurrency control is not enforced. Consider the transaction of Figure 8.5 that adds a certain amount to a given account. Consider now that the target account "my_account" has an initial value of zero and that two concurrent transactions T1 and T2 execute `deposit(my_account,10)` and `deposit(my_account,20)` respectively. If transactions are executed one at a time, the final result will be 30 in `my_account` has you would probably expect. However, we want transactions to execute concurrently, but unfortunately, the interleaving of instructions illustrated by Figure 8.6 results in a final value of 20, as if only one of the transactions was executed. This particular problem is known as the "lost update" problem.

```
T1 x1 = my_account.read (); // reads 0
T2 x2 = my_account.read (); // reads 0
T1 x1 = x1 + 10;      // x1 = 10
T2 x2 = x1 + 20;      // x2 = 20
T1 my_account.write (x1); // my_account = 10
T1 my_account.write (x2); // my_account = 20
```

Figure 8.6. Lost Update Problem

Other similar problems can occur. Consider the same example. A transaction withdraws a given amount from one account and deposits the same amount in another account. If another transaction tries to compute the sum of both accounts, and accesses the first account *after* the withdrawal and the second account *before* the deposit has been made it will find that some money is missing. This problem is known as the *inconsistent retrieval* problem.

Enough for the problems! Something needs to be done to prevent these scenarios from occurring, and this something is called concurrency control. The goal should be to allow as much concurrency as possible. Of course, we could prevent the concurrent execution of transactions by using a global lock on all data (for instance, on the complete database). The resulting performance of the system would be worse than deplorable. Typically, many transactions access unrelated data items and can be executed concurrently without any type of restrictions. What is needed is a mechanism that allows transactions to execute concurrently as long as they produce the same results has if they were executed in (some) serial order. This correctness criterion is known as *serializability*.

A huge body of theory exists on concurrency control for databases and, in particular, on enforcing serializability. Here we will just address one of the simplest (and popular) mechanisms: *locking*. Locking uses two types of locks: *shared* locks and *exclusive* locks. Whenever a transaction accesses a data item it locks that item: if the transaction accesses the item for reading it acquires a shared lock (other read locks can be granted); if the transaction accesses the item for writing it acquires an exclusive lock (no one else can grab this item). Only shared locks are compatible. If at least one of the locks is an exclusive lock the locks are incompatible, as illustrated in Table 8.1. If the item is already locked by another transaction, the transaction must check first if the locks are compatible; if they are not compatible, the transaction must wait until the previous lock is released in order to acquire its lock.

Locks cannot be released as soon as the operation completes; this would be almost as good as not having locks at all (just try to add an acquire immediately before and a release immediately after every item access on Figure 8.6). It has been shown that if a transaction does not release any lock before acquiring all the locks it needs, serializability can be enforced. This is known as *two-phase locking* (2PL), because the locks are first acquired (this phase is also called the *growing* phase) and later released (this phase is called the *shrinking* phase).

Table 8.1. Lock Compatibility

Lock Types	Shared	Exclusive
shared	compatible	incompatible
exclusive	incompatible	incompatible

We have already noted that if a result from a transaction that later aborts is seen by another transaction, that second transaction also needs to abort. This rule can easily be applied in a recursive manner. If a third transaction sees an intermediate result from the second transaction, the third transaction will have to abort too. This phenomenon is known as *cascading aborts* or sometimes, by the more visual name of *domino effect.* Cascading aborts are not a positive feature, since they require work to be undone and this means poor resource usage. A way to prevent cascading aborts is to prevent intermediate results from being visible before the transaction terminates. When locking is used, this can be achieved simply by holding all the locks until the transaction commits or aborts; this restriction to the two-phase locking rule is so popular that deserves a name of its own: *strict two-phase locking.*

Locking is simple but not exempt from disadvantages. One of the problems with locking is that it may cause *deadlocks.* Consider for example a transaction T1 that transfers an amount from account A to account B and another transaction T2 that transfer an amount from account B to account A. If T1 locks the account A and T2 locks the account B, none of the transactions will be able to make progress. Many different strategies can be used to prevent, avoid and detect deadlocks (*see Coordination* in Chapter 2). The most simple strategy is to define a maximum waiting time for a lock to be released. If this timeout expires a deadlock is assumed and the transaction is aborted.

8.6.3 Distributed Transactions

Distributed transactions are simply transactions that access items on different nodes of a distributed system. Distributed transactions can be built using the mechanisms developed for centralized transactions. A generic block diagram of a distributed transaction system architecture is depicted in Figure 8.7, showing the interaction between all TM modules, which competitively launch transactions, and local Scheduler modules, which must ensure that the distributed transaction is scheduled correctly at every site.

In fact, a distributed transaction can be described as a collection of several sub-transactions, each individual sub-transaction being initiated on each node visited by the distributed transaction. The local transactional mechanisms on each site will guarantee that each individual sub-transaction either executes completely or aborts. Since locking is a *local* concurrency control mechanism, i.e., locks are associated with individual data items, it can be used to enforce serializability of distributed transactions.

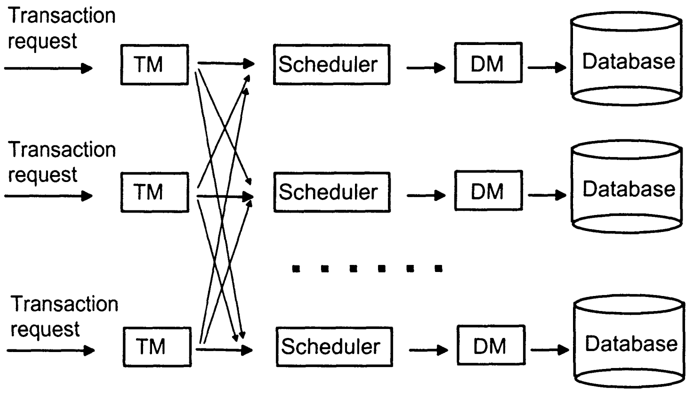


Figure 8.7. Distributed Transaction System Architecture

Distributed transactions have nevertheless a unique feature that is not present in centralized transactions. All the participating nodes need to agree on the outcome of the transaction. Thus, nodes participating in a distributed transaction need to execute an *atomic commitment* protocol. These protocols have been discussed earlier (*see Recovery* in Chapter 7).

8.6.4 Transactions and Replicated Data

Replication can improve the availability of data and can also improve the system performance, by putting data closer (in terms of communication delays) to its user. A transaction system that supports access to replicated data should support replication transparency. This means that the several replicas of the data should look just like a single copy to their users. This is called *one-copy equivalence* and the corresponding correctness criteria is called *one-copy serializability*.

One intuitive way of making several replicas look like a single copy is to keep all the replicas with exactly the same value. In other words, when one replica is updated, all replicas are updated. Since all the replicas always have the same value, when a read is performed you just need to read a single replica (this is also called the *write-all, read-one* approach). At this point, you probably recall the state machine of Section 7.6.2 as a general technique to maintain consistent replicas. We will now ask you to put aside the state-machine approach for some moments, and to think of how to maintain replica using exclusively transactional mechanisms. In terms of transactions, updating all the replicas means including the writes to different replicas within the same transaction. Thanks to the all-or-nothing property of transactions, this will ensure that either all replicas are updated (if the transaction commits) or none is (the transaction is aborted).

In a system where replicas never fail, the write-all approach works fine, at least from a consistency point of view. Of course, the performance of this approach is highly dependent on the number of replicas and on the communication efficiency. However, if a replica fails the system comes to a halt, since not all replicas can be updated. Fortunately, we have already presented solutions for this problem (*see Replication Management in Partitionable Networks* in Chapter 7). Let us just recall some of the paradigms that can be applied to this problem: weighted voting, coteries, structural representations and dynamic voting. These solutions will preserve one-copy serializability even in the presence of partitions. Note that write quorums should always intersect among themselves and with read-quorums. This means that conflicting operations always intersect in at least one replica. This replica can detect the conflict using a local concurrency-control mechanism (such as locking).

Suppose now that you are in an environment where network partitions (real or virtual) are very unlikely. In such cases, when a replica does not respond this means that the replica is crashed. Voting can be too penalizing, since in order to increase the availability of writes it reduces the performance of reads (which can no longer be done from a single replica). Why not just try to keep all available copies updated and forget about the crashed replicas? This approach is known as *write-all-available* copies approach. Of course, one has to be careful with recovering replicas because these replicas are out-of-date. Thus, recovered replicas cannot be read until they are brought up-to-date.

Does the available replicas approach really work? The answer is yes if special care is taken in dealing with crashes. Consider the following scenario. There are two data items A and B with different initial values, say $A = 50$ and $B = 100$. Consider that a transaction T_1 sets $B := A$ and a transaction T_2 sets $A := B$. The final outcome depends on the order by which T_1 and T_2 are serialized. If T_1 is serialized before T_2 both A and B will be set to 50. If T_2 is serialized before T_1 both A and B will be set to 100. In any case, the final value of A should be the same of B. Suppose now that there are two copies of each account, denoted A_1, A_2, B_1 and B_2 and that A_1 and B_2 fail during the concurrent execution of T_1 and T_2. Now, let us look at what happens if transactions execute as follows:

1. *Transaction T_1 reads A_1*
2. *Transaction T_2 reads B_2*
3. *A_1 and B_2 crash*
4. *Transaction T_1 updates just B_1 because B_2 has crashed*
5. *Transaction T_2 updates just A_2 because A_1 has crashed*

The final amount of the remaining replicas will be different (in fact, this execution swaps the values of A and B)! What went wrong? The problem is that the transactions were operating with different sets of available replicas: T_1 did read A_1 but T_2 only wrote A_2. In order for available replicas to work properly, crashes need to be serialized with data accesses. This should not come as a surprise since we have already discussed the concept on virtual synchrony (*see Consistency* in Chapter 7) and the relevance of ordering failure information with regard to application messages.

This brings us back to the state-machine approach. Are transactions and state machines incompatible methods of managing replicated data? Not really. Actually, they can be seen as complementary methods. Transactions guarantee the atomicity of sequences of operations; this is an aspect that is not addressed by the state-machine approach. Replicated state machines put emphasis on replica consistency; although transactional mechanisms can also address this issue they can be improved with the lessons learned from building state machines.

Imagine that two concurrent transactions and T_1 and T_2 try to lock a replicated data item. Unless special care is taken, the lock requests may arrive in different orders to the two replicas. Thus, one replica can be locked by T_1 and the other by T_2. This would result in deadlock. The replicated state-machine solution to this problem would be to use an atomic multicast primitive, ensuring that both replicas receive the same lock request in the same order, preventing the deadlock from occurring. Recent work has shown that the use of ordered group communication primitives can improve the performance of replicated database management systems (Pedone et al., 1998; Kemme et al., 1999).

8.6.5 Building Transactional Systems

Systems where the users submit transactions and wait for the outcome of the transaction are called *On-Line Transaction Processing* systems (OLTP for short). Transactional systems can be used to build OLTP applications but also to build applications that operate off line or in batch mode.

An example of a batch transaction processing system is the system that processes check payments. Checks issued by bank A and deposited in bank B during the day are sent in batch to the issuer at the end of the day (usually, through some third party clearinghouse to avoid the need for each bank to contact every other bank directly). Bank A will then debit the appropriate accounts and/or register exceptions such as lack of funds. Again, the results of the transactions are sent back in batch to bank A (usually, one or more days after) which in turn will credit the accounts where the checks were deposited.

On the other hand, some operations performed with a debit card on an automatic teller machine require OLTP support. Operations such as reading the account balance require the execution of a distribution transaction that, ultimately, needs to contact the computing system of the issuing bank. Building OLTP systems raises many challenges which are not limited to fault tolerance. These systems are usually very large in terms of users and volume of data and must be able to withstand a very large number of transactions per second. This requires a clever design in terms of operating system constructs (how the transactional system is decomposed into processes and threads), memory management (the hierarchy from disk to CPU cache) and communications (to ensure that enough bandwidth is available). Obviously, these systems also pose enormous challenges in terms of security but these issues will be dealt with in Part IV of this book.

Disaster recovery is often considered as a must-do item in a large enterprise information system. The idea is to have contingency plans, should a major disaster occur that would severely degrade the operational capability of the information system, namely its public presence, mainly through the OLTP operations. However, there is a misunderstanding: what disasters are we talking about— informatics[2] disasters (computer blowing-up, disks failing, backups lost)? Or *real* disasters (major floods, massive power outages, large-scale fires, earthquakes)?

Many companies devote (expensive) contingency plans to both kinds. However, it seems that the first ones can and should be avoided by technical, run-time measures. Fault tolerance is failure avoidance. Tolerance of faults that can yield catastrophic effects is ***disaster avoidance.*** Distributed fault tolerance is just about the paradigm to handle serious faults that can have a geographical dependency. In reality, it only makes sense to make contingency plans for the events that cannot be avoided, such as environmental disasters with global proportions. Very often, it is better to prevent than to remedy.

8.7 SUMMARY AND FURTHER READING

This chapter discussed how to apply the paradigms presented in Chapter 7 to build fault-tolerant systems. It departs from a systematization of failure classes and how these impact the approaches to fault tolerance, both in terms of techniques chosen and in terms of the service provided. Detailed evaluation of failure assumption coverage of hardware and software components can be found in (Iyer and Joshi, 1985; Madeira and Silva, 1994; Arlat et al., 1990; Carreira et al., 1998; Maxion and Olszewski, 1998). See (Kopetz et al., 1989a; Powell, 1991) for systems built to the strong and weak fail-silence assumptions.

Then, we have focused on three of the major constructs to build reliable distributed applications: remote operations, event based systems and transaction systems. We discussed the application of the paradigms discussed eralier to build these types of systems. Additional readings in reliable remote invocation systems can be found in (Cooper, 1985; Liskov et al., 1987; Panzieri and Shrivastava, 1988; Rodrigues et al., 1994). Fault-tolerant event-channels have been presented in (Oki et al., 1993; Felber et al., 1997). Naturally, there is a huge bibliography on transactions, from which two books emerge as fundamental references: (Bernstein et al., 1987; Gray and Reuter, 1993).

[2] "Informatics" is a word of european origin getting increased acceptance in the community of computer users and developers. It is used to denote in general terms all that is related with use of computers and networks in information processing, access and manipulation.

9 DEPENDABLE SYSTEMS AND PLATFORMS

This chapter gives some examples of dependable systems and platforms. Whenever possible, paradigms and models previously studied are pointed to the reader. The overview of each system is concise and the selection is subjective but tries to illustrate each class of approaches using concrete case studies that we find representative of that class. Namely, we discuss: distributed fault-tolerant systems, transactional systems, cluster architectures, and how to make legacy systems dependable. In each section, we will mention several examples in a summarized form, and then will describe one or two the most relevant in detail. Table 9.1 at the end of the chapter gives a few URL pointers to where information about most of these systems can be found.

9.1 DISTRIBUTED FAULT-TOLERANT SYSTEMS

9.1.1 Tandem and Stratus

Tandem (now belonging to Compaq) and Stratus are two companies that have specialized in building fault-tolerant systems. Solutions offered by these companies include a combination of hardware and software fault tolerance that offer reliability and continuous availability. The products that achieve higher degree of reliability use proprietary hardware and software (including specialized operating systems). We will focus on some of their most innovative architectures.

Stratus was the first company to introduce a processor architecture that used duplication at the level of CPU, I/O and communication hardware. In this architecture, a CPU server with crash failure semantics is obtained using a pair of microprocessors that execute in lock-step and whose results are compared. To offer availability, CPU servers are replicated using the state-machine approach; in this way a pair of CPU servers may mask the crash of single server. At the memory level, a two-bit detection/ one-bit correction coding is used to build a memory unit with crash failure semantics.

Tandem developed a set of products for high-reliability transaction processing called the Guardian 90 system. The set includes a single operating system that offers transactions in the kernel and supports the parallel execution of *process pairs* on duplicated hardware. In the process pair approach, each active process has a backup process associated to it. All actions of the active process are checkpointed to the backup such that, in the case of failure of the active, the backup can continue the operation. In terms of hardware, the Guardian executes on top of a fully duplicated architecture. The hardware is configured to ensure the existence of two disjoint paths from terminals to servers. Today, Tandem offers a wide range of solutions combining the process-pair approach and hardware redundancy (lock-stepped microprocessors).

The pressure for competitive solutions even if offering lower levels of reliability lead many companies to explore an alternative track, applying their expertise to enhance commercial off-the-shelf components (COTS), using modular (software-based) fault tolerance techniques. Target markets are high-reliability, high-availability versions of servers based on commercial operating system "standards" such as Unix and WindowsNT/2K.

9.1.2 The Isis family

Isis (Birman and van Renesse, 1994) started as a research project at Cornell University that focused on the use of process groups to build reliable distributed applications. The core of the Isis system was an innovative set of group membership and reliable communication primitives, enforcing different ordering properties including causal and totally ordered multicast. Isis was also the first system to introduce the important concept of virtual synchrony. The main services offered by ISIS were: ISIS Toolkit (basic protocols); Reliable Distributed Objects (object interface to groups); Distributed News (a publisher-subscriber facility); Reliable NFS (non-stop NFS); Distributed Resource Manager (load balancing and coarse-grain distributed parallelism). The architecture of ISIS is depicted in Figure 9.1.

Isis was later supported commercially by a startup company called Isis Distributed Systems (IDS), later acquired by Stratus. The combination of a professional support and a competitive pricing to universities helped to disseminate the project results and made Isis a reference system in the area of group communication.

The launch of IDS did not terminate the research on group communication at Cornell. Different generations of protocols that improved previous work have been developed. The design of Isis involved with the experience gained with the usage of the system in large-scale "real-life" scenarios. Eventually, the system

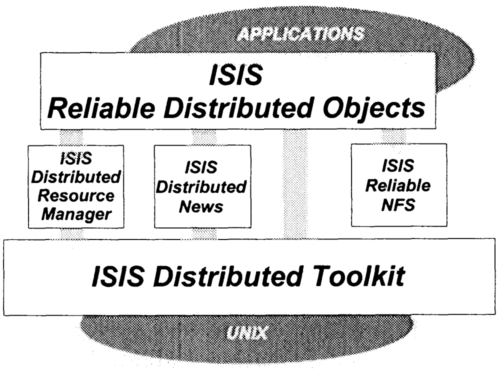


Figure 9.1. ISIS Architecture

was completely redesigned, leading to the development of Horus (van Renesse et al., 1996), a system with the same aims but with a completely different architecture, with emphasis on modularity, protocol composition, operation in partitionable environments, and security. The latest generation of the system is called Ensemble, a group communication system written in the ML programming language with a number of innovative features, such as support for formal validation of protocols (Hayden, 1998).

9.1.3 Arjuna

Arjuna (Shrivastava et al., 1991) is a distributed fault-tolerant system that integrates several software-based fault tolerance techniques. Originally developed at the University of Newcastle upon Tyne (UK), it is now commercially supported by Arjuna Solutions Ltd. Arjuna is a distributed platform, consisting of libraries, stub compilers, run-time mechanisms and services (protocols, servers) that support the development of fault-tolerant applications using programming languages such as C++ and Java. The project was started in 1987 and the first public release of Arjuna code was distributed in 1991.

Maybe the most important fault-tolerant mechanism supported by Arjuna is nested transactions. Using Arjuna, the programmer can build objects that are persistent and distributed. Object methods are invoked using remote procedure calls in the context of transactions. The system manages the state of the object, bringing the object to main memory (what is called activating the object) and later saving its new state when the transaction commits. Concurrency control is managed by Arjuna to ensure that the execution of concurrent transactions is serializable. Today, the system includes a CORBA Object Transaction Service. Object replication was also added to Arjuna, allowing different replicas of a given object to be maintained in different servers. To simplify the management of replica consistency, a multicast communication layer and group management services were added to the system.

Arjuna is an example of a research project that was able to incorporate ideas from previous transactional systems, such as Argus (Liskov, 1985) and Avalon (Eppinger and Spector, 1991), and group oriented systems such as Isis (Birman and Joseph, 1987) in a comprehensive prototype that reached the maturity to migrate to the commercial arena.

9.1.4 Delta-4

The Delta-4 (Powell, 1991) architecture was aimed at the development of fault-tolerant distributed systems, offering a set of support services implemented using a group-oriented approach. An object-oriented application support environment provides separation of concerns: it allows building applications with incremental levels of fault tolerance, while the non-functional properties concerned with achieving dependability are secured transparently from the applications programmer. Black-box commercial applications can also be rendered fault-tolerant without change, through special transformer objects (wrappers). The architecture was designed and developed by a consortium of several European corporations, institutes and universities, under the ESPRIT research programme. Delta-4 stands for "**D**efinition and **D**esign of an Open **D**ependable **D**istributed Architecture". To the authors' knowledge, Delta-4 was one of the first architectures to integrate, at all levels, modular and distributed fault tolerance concepts, namely relying on the emerging group communication and membership technologies. An excellent perspective on Delta-4 is offered by Powell (Powell, 1994).

A Delta-4 system consists of a number of computers (possibly heterogeneous) connected by a reliable communications system. The application programs consist of software components distributed among the system's nodes. A given component may be replicated, its copies being executed on different machines. Each machine consists of a *hosting node* and a *Network Attachment Controller* (NAC). NACs are dedicated communication boards where a reliable communication system is executed, supporting reliable multicast communication among computational entities. The NACs are the single hardware component specific of the Delta-4 architecture: collectively, they implement a reliable group communication abstraction under a fail-silent assumption. On top of it, hosting nodes may have any behavior, even fail-uncontrolled (i.e., arbitrary). The combination of the hosting node with the NAC forms a network node. The Local Executives[1] (LEXs) of the host machines can also be heterogeneous. Each NAC runs a real-time kernel as the execution environment. The host may run any operating system, e.g., UNIX. It may also run a real-time O.S. in the real time version of Delta-4, the XPA (*see Dynamic Systems* in Chapter 14).

The modular architecture of Delta-4 is represented in Figure 9.2. The distributed software running on the nodes can be classified as follows:

[1]Name given to the local execution environment, usually an operating system or a real-time kernel.

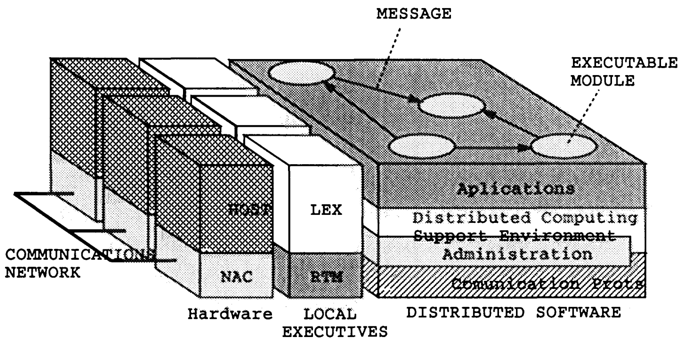


Figure 9.2. Delta-4 Architecture

- The communications software executed on the NACs; the communications system of the architecture is named the *Multicast Communication System*, offering multi-point reliable connections.
- The system administration software, managing the computational elements (including the communication system), which is executed partially in the host computer and partially in the NAC.
- The *Applications Support Environment*, named DELTASE, that supports the development of distributed applications. DELTASE was designed according to emergent standards such as the ODP model[2] (ODP, 1987) and closely followed the work of other organizations involved in the standardization process, namely the ANSA (ANSA, 1987) architecture[3].
- The user software, composed of several components, potentially developed using different programming languages, that interact through *Remote Service Requests.*

The *Multicast Communications System* (MCS) was designed using an OSI[4] like layered architecture. Delta4-XPA used a simplified high-performance version (*see Dynamic Systems* in Chapter 14). The main layers of the communication architecture are:

- *Membership and Multicast* local-area communication.
- *Reliable Transport Service*, offering reliable multi-point connections.
- *Replication Management*, a key element for implementing the fault tolerance techniques of the architecture. This level coordinates the communication among replicated access points, guaranteeing the message delivered to all addressed points and filtering duplicated messages. This component can execute error detection and error compensation protocols when needed.

[2]Open Distributed Processing.
[3]Advanced Networked Systems Architecture.
[4]Open Systems Interconnection.

- *Session* and *Presentation* multi-point services.

The services of the Multicast Communication System are used to support three fault tolerance techniques: Active replication; Passive replication; Semi-Active replication.

Active replication as implemented by MCS supports both fail-silent and fail-uncontrolled hosts, by providing non-voting or voting algorithms, respectively. Failure detection is also passed on to System Administration (for example, for cloning the replica in another node). In the second technique, *passive replication*, a component may be replicated but only one copy remains active, periodically sending a snapshot of its own state to the remaining copies. This approach can only be applied to fail-silent components. Finally, *semi-active replication* leaves the initiative to a designated replica, the *leader*, which processes input messages, generates replies, and instructs the *followers* to execute the same steps in the same order, without producing outputs. This technique allows fast error recovery, and accommodates some non-determinism not allowed by active replication.

In the Delta-4 architecture, the complexity of all fault-tolerant mechanisms is hidden from the application programmer by the support environment DELTASE, in what constituted a pioneering object-oriented transparently fault-tolerant middleware, featuring characteristics such as the provision of incremental and configurable degrees of replication and fault resilience. On the other hand, these techniques are complemented by a powerful system administration component, responsible for diagnosing system faults and triggering the appropriate fault treatment operations, such as automatic reconfiguration, cloning, and so forth.

9.1.5 The Information Bus

The Information Bus (TIB) (Oki et al., 1993) is a commercial product developed by Teknekron Sotfware Systems, Inc, a company co-founded by Dale Skeen. It consists of a distributed environment for the development of fault-tolerant applications based on the publisher-subscriber model.

The Information Bus supports an event-driven communication model, where publishers inject data objects on the bus tagged with a *subject* string. Subscribers register the subjects they are interested in with the system, and subsequently receive a copy of data objects associated with these subjects. Events can be published with different quality-of-service requirements. The weaker semantics is *reliable* delivery, which guarantees that messages are delivered only once in FIFO order as long as the network does not suffers a partition and both the publisher and the subscriber do not crash. The stronger *guaranteed* delivery ensures delivery regardless of failures; in this case the message is logged in non-volatile storage before being sent and retransmitted until a reply is received.

The Information Bus also offers an RPC mechanism with *at-most-once* semantics in the presence of failures. To establish the binding between the client and the server, the underlying publisher-subscriber service is used. The client publishes an inquiry searching for servers. The relevant servers publish an advertisement with address information to allow the client to establish a point-to-point connection.

In order to integrate legacy applications with the Information Bus, one has to build dedicated software modules, called *adapters*, which are able to convert the data objects used in the Bus into representations understood by those applications. Today, Teknekron Software Systems is called TIBCO and offers different products that exploit the concept of reliable publish-subscribe interaction.

9.2 TRANSACTIONAL SYSTEMS

9.2.1 CICS

The Customer Information Control System (CICS), was originally developed by IBM in 1968 to support the interaction of terminals with mainframe computers. In the original CICS model, each terminal sends input messages to a server on the mainframe, which invokes a program to process the message. The server was actually a single operating system process in whose address space both CICS and all its applications execute. Making CICS execute in user space, along with that fact that CICS makes no use of peculiar operating system features, made the system extremely portable. On the other hand, executing all services in the same address space also made the system very vulnerable to software faults in any of the involved applications.

CICS integrates a wide range of services, including presentation services (that take care of data-format translations), session services (that allows programs to open, and later close, channels to other programs or devices), storage services (with different semantics), file services, transaction management, journal management, recovery management (for transaction abort, shutdown, restart and recovery from tape), program management (linking, loading and execution), thread management, authorization and authentication.

Distributed transaction processing is also supported by CICS since transaction submitted to a given CICS system can be routed to another CICS system or invoke remote operations on other systems. The interacting CICS systems can be located on the same machine or in different machines. The reason for having more than one CICS in the same machine is to improve fault-containment. In this case, remote invocations are performed efficiently using shared memory. If CICS are on different machines, communication can be performed using IBM's transactional communication service LU6.2 which offers several qualities of service, the stronger of which supports ACID properties across distributed transactions.

9.2.2 Encina

Encina is an OLTP System originally built by Transarc, a company founded in 1989 by Alfred Spector. Encina is an evolution of previous work by Spector at Carnegie Mellon University on transactional systems (Eppinger and Spector, 1991) and was designed in the framework of the Distributed Computing Environment (DCE) of the Open Software Foundation. Encina uses other DCE services as building blocks, such as the DCE RPC for implementing remote procedure calls and Kerberos for authentication and encryption. Transarc became a subsidiary of IBM in 1994, and Encina is now part of the IBM's transaction processing products, TXSeries[tm], that also includes CICS and the distributed file system AFS.

To simplify the task of building distributed applications with transactional semantics using the C programming language, Encina includes a specialized programming environment called Transactional-C. This environment includes a number of extensions to the C programming language, such as directives to express concurrency (supported in run-time by a thread package) allowing C programmers to launch concurrent sub-transactions in an elegant way. With the advent of the OMG architecture, Encina was enhanced to support the development of C++ servers and C++ or Java clients running on the Orbix object request broker.

9.3 CLUSTER ARCHITECTURES

Cluster is a designation that has been used to describe a family of systems with quite diverse characteristics. In general terms, a cluster is an integrated collection of machines that, to some extent, can be managed has a whole and, sometimes, can be interfaced by the external nodes as a single machine. Clusters have been built to offer high processing power (as a cheap alternative to expensive specialized multiprocessor architectures) or to offer higher availability, usually by providing a convenient way to restart a service on a different cluster node when a crash occurs. Most hardware and software vendors have cluster products, including Sun, Compac, HP, IBM, Microsoft, etc. A complete book can be written and has been written just about clusters (Pfister, 1998) so we will concentrate on features related to fault tolerance.

The key aspect of clusters, from the availability point-of-view, is that they are made of several processing nodes and several storage nodes interconnected by some high speed link. It is possible to have each storage node statically assigned to each processing node. However, storage nodes are frequently shared among processing nodes, either by making the storage nodes interface the network or by using dual-port disks. These alternatives are illustrated in Figure 9.3. Alternative (c) has received enormous interest recently, given the need for huge amounts of storage to be offered in a shared and distributed way. Products emerging under this technology are called *Storage Area Networks (SAN)*.

It is possible to develop applications that make use of the existing redundancy to run in non-stop or reconfigurable modes. Examples of the latter are

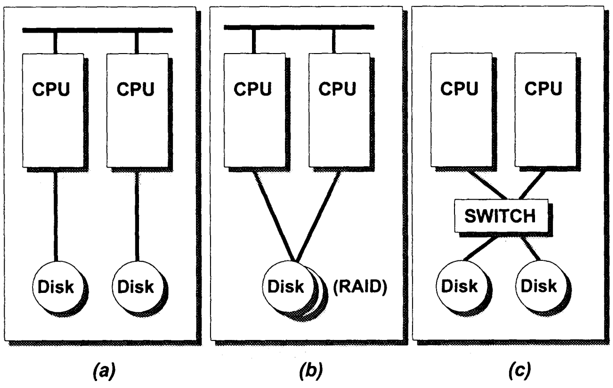


Figure 9.3. Storage in Cluster Architectures: (a) Non-shared; (b) Local sharing; (c) Network sharing

fault-tolerant versions of file systems, such as the Highly Available NFS (HA-NFS) (Bhide et al., 1991), or of database servers such as the ones available from vendors such as Oracle or Informix. However, clusters are often used in recoverable mode. Nodes are used for load sharing, individually running "standard" applications unaware of cluster facilities. The cluster software allows the application to be launched on any computing node. If a crash occurs, the application goes down, and cluster management facilities are able to launch the application in another node of the cluster.

Services provided by the cluster software and hardware usually include failure detection services, reliable communication services, event managers that are able to provide each member of the cluster with information about the cluster status (for instance, to distribute load information), configuration managers (where administrators may specify dependencies among applications and location policies), load balancing managers, fail-over managers, etc. With regard to failure detection it is worth noting that the use of dedicated networks to connect the cluster nodes increases the synchrony of the system and, therefore, renders failure detection easier. Nevertheless, most clusters that use a primary-backup approach exploit specific hardware interfaces, like the SCSI challenge-defense protocol, to ensure that two nodes cannot simultaneously consider themselves as primary.

9.4 MAKING LEGACY SYSTEMS DEPENDABLE

Most of the examples described in the previous sections used the approach of constructing a fault-tolerant system or toolkit from scratch. Clearly, this is the most effective approach to optimize the efficiency of solutions. However, the system architect is often challenged with the task of making legacy systems dependable without re-implementing existing applications (a task that may not be feasible because of timing or cost constraints).

One of the most successful approaches to deal with legacy systems consists in building a new interface to the legacy component. The new interface is responsible for intercepting inputs and outputs and manipulating these interactions to make them suitable to implement a given fault-tolerant strategy. Consider for instance that you have a legacy system that behaves like a state machine. In this case, the system can be made fault-tolerant by replicating the component. This means that the interface must intercept the inputs and multicast them in total order to all replicas. The outputs must also be intercepted and combined in a single output. The approach is illustrated in Figure 9.4.

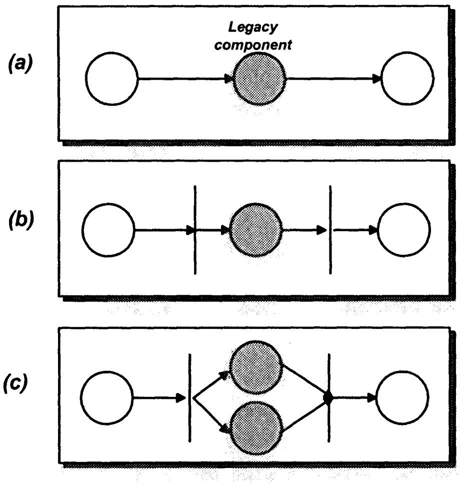


Figure 9.4. Interfacing Legacy Systems

This approach has been used successfully in many systems under different names, such as *transformers* in the Delta-4 project, *adapters* in the TIB approach, *wrappers* (Birman, 1996), or *metaobjects* (Fabre et al., 1995). There are many ways of implementing wrappers. If the component interacts with other components through message passing, it obviously becomes easier to make the interposition. If all interactions are performed using some standard RPC-like mechanism it becomes even simpler. For instance, CORBA (OMG, 1997a) components can be wrapped by generic components using dynamic invocation facilities (Felber et al., 1996). Otherwise some other operating system or language-specific techniques may be used. For instance, specialized dynamically linked libraries may be provided to intercept all the system calls made by a given component. Alternatively, specialized language libraries or specialized language run-time environments can be provided to support the component execution.

It should be noted that sometimes it is feasible to alter the way the clients interact with the servers. In such case fault-tolerant features can be embedded in the client protocol, for instance enhancing clients with the ability to multicast requests to a group of servers. At first glance this may look a bit intrusive but there is a general tendency to make legacy applications available through new interfaces, such as web browsers, even when no fault tolerance concerns are involved. Additionally, technologies such as Jini (Waldo, 1999) allow the client to dynamically download the right interface from the name server.

On the other hand, if it is not affordable to change clients, the wrapper must make the fault-tolerant features fully transparent for the client protocol. Consider for instance the case where the client uses the UDP protocol to communicate with the server and that a primary-backup scheme is selected to make the server fault-tolerant. In case of failure of the primary, the backup must be able to impersonate the IP address of the primary such that the client continues to use exactly the same address as before. Of course, migrating an TCP connection without disruption requires a bit more work.

9.5 SUMMARY AND FURTHER READING

This chapter has given concrete examples of research and commercial fault-tolerant systems. The number of systems cited was necessarily small and many excellent systems were not mentioned. For access to downloadable software or further reading, Table 9.1 gives a few pointers to information about some of the systems described in this chapter.

In terms of fault-tolerant systems coming from industry, documentation on the concepts underlying the Advanced Automation System prototype (Cristian, 1994; Cristian et al., 1996) and the Parallel Sysplex Cluster (Bowen et al., 1997a; Bowen et al., 1997b) are definitely worth reading. For detailed material on cluster architectures, see (Pfister, 1998).

Group communication systems have been a field of very reach research. In addition to the work at Cornell, many other groups have developed long and solid work on the area. We list some of the most relevant systems: Transis (Amir et al., 1993a), Totem (Moser et al., 1995) and the associated concept of extended virtual synchrony (Moser et al., 1994), Psync (Peterson et al., 1989) and Consul (Mishra et al., 1993), NavTech (Rodrigues and Veríssimo, 1995; Rodrigues et al., 1996), Relacs (Babaoğlu et al., 1994) and RMP (Callahan and Montgomery, 1996).

For further work on object-oriented fault-tolerant middleware, see (Fabre et al., 1995). Other relevant systems are Manetho (Elnozahy and Zwaenepoel, 1992b; Elnozahy and Zwaenepoel, 1992a) and Harp (Liskov et al., 1992). One of the pioneer works on transaction systems was Argus (Liskov, 1985) developed at MIT by Barbara Liskov, whose most recent creation is THOR (Liskov et al., 1999). Fault-tolerant transactional systems based on coordinated atomic actions are described in (Xu et al., 1999).

Table 9.1. Pointers to Information about Fault Tolerant Systems and Platforms

Class of System	*System*	*Pointers*
IETF RFCs		www.rfc-editor.org
OMG	(FT-CORBA)	www.omg.org
IEEE-IFIP	(Dependab.)	www.dependability.org
Message Buses		
	TIBCO	www.tibco.com/
	iBus	www.softwired-inc.com/
Group Communication	**ISIS**	www.cs.cornell.edu/Info/Projects/ISIS
	HORUS	www.cs.cornell.edu/Info/Projects/Horus
	Ensemble	www.cs.cornell.edu/Info/Projects/Ensemble
	Relacs	www.cs.unibo.it/projects/relacs
	Transis	www.cs.huji.ac.il/labs/transis
	Totem	beta.ece.ucsb.edu/totem.html
	Phoenix	lsewww.epfl.ch/projets/phoenix
	x**AMp**	www.navigators.di.fc.ul.pt/
	Spread	www.spread.org
Checkpointing		
	Manetho	www.cs.cmu.edu/˜mootaz/manetho.html
	Libckpt	www.cs.utk.edu/˜plank/plank/www/libckpt.html
	Egida	www.cs.utexas.edu/users/lorenzo/lft.html
Transactional systems	**Argus**	www.pmg.lcs.mit.edu
	THOR	www.pmg.lcs.mit.edu/Thor.html
	Arjuna	arjuna.ncl.ac.uk
Fault-tolerant Systems	**Eternal**	beta.ece.ucsb.edu/eternal/Eternal.html
	Electra	www.softwired-inc.com/people/maffeis/electra.html
	Cactus	www.cs.arizona.edu/cactus
	OGS	lsewww.epfl.ch/OGS
	Filterfresh	www1.bell-labs.com/org/11356/
Validation and Verification	**LAAS**	www.laas.fr
	Critical	www.criticalsoftware.com
	Ballista	www.cs.cmu.edu/˜koopman/ballista
	ULTRASAN	www.crhc.uiuc.edu/PERFORM
Tools	**Orchestra**	www.eecs.umich.edu/RTCL/projects/orchestra
	Kronos	www-verimag.imag.fr
	PVS	pvs.csl.sri.com
Clusters and Commercial Platforms	**IBM**	www.research.ibm.com
	TANDEM	www.tandem.com
	STRATUS	www.stratus.com
	Microsoft	research.microsoft.com
	Compaq	www.compaq.com/enterprise/highavailability.html

10 CASE STUDY: MAKING VP'63 DEPENDABLE

This chapter takes the next step in our case study: making the VP'63 (VintagePort'63) Large-Scale Information System dependable. Increased reliance on computers for day-to-day operation on the one hand, and greater geographical dispersion of the system on the other, have raised concerns about the impact of service outages or even severe failures on the business results. In consequence, part of the study concerns the enhancement of the reliability and availability of the VP'63 system.

10.1 FIRST STEPS TOWARDS FAULT TOLERANCE

The reader should recall that this is the next step of a project implementing a strategic plan for the modernization of VP'63, started in Chapter 5, and continued in the Case-Study *chapters of each part of this book. The reader may wish to review the previous part, in order to get in context with the project.*

The development strategy laid out will impose a growing dependence of the business on the computing infrastructure. The most important facet of the desired dependability of the services provided by VP'63 is thus availability. Whilst it is desirable that the system does not fail often (reliability), it should also exhibit small glitches, remaining operational for a very high percentage of its life time.

The client-server front-end to the database is already fragmented. This is a first step at independence of failure, as recalled in Figure 10.1a: local transactions will not be affected by the failure of servers of the other fragments. However this is not enough to guarantee overall availability, since operation related with that fragment stalls.

On the other hand, the publisher-subscriber infrastructure is based on a single-location server (Lisboa) which, once down, stalls the whole service, as depicted in Figure 10.1b. This is a severe availability impairment.

The 3-tier DFS-Web architecture has already a few aspects enhancing availability: RO file or volume replicas are accessible by any client, and so the initial objective of performance also serves availability, since upon failure of the local RO copy, the client can fetch a remote one. This may eventually be extended to replication of RW files or volumes, with a DFS that support this feature.

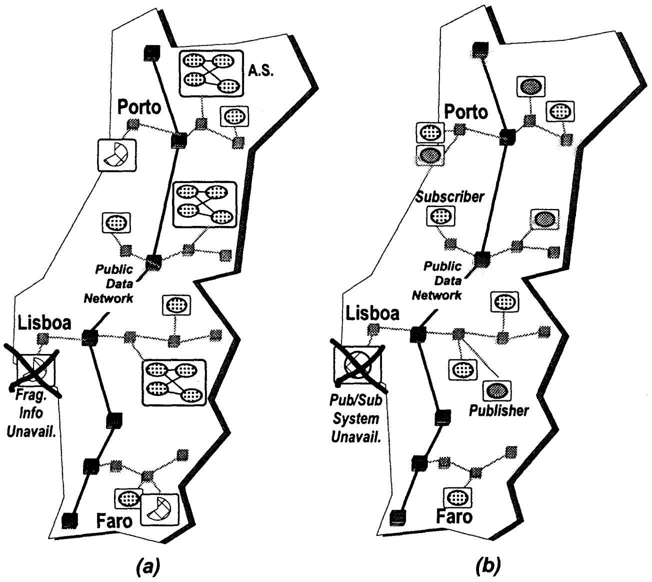


Figure 10.1. Failure Scenarios: (a) Fragmented DB; (b) Pub-Sub System

10.2 FAULT-TOLERANT CLIENT-SERVER DATABASE

Q.2. 1 What can be done to improve the availability of the database server?

One possible answer is to go for a replicated distributed database server. All fragments are now replicated in other sites. The team decided to use a level of replication of $n = 2$ for the first prototype, since in normal environments, with a fair maintainability, known statistics show that availabilities in excess of two nines can be achieved. Since the database is fragmented by enough sites (*see* the initial situation in Figure 5.3a), it is not necessary to allocate extra machines to achieve fault tolerance.

The modular fault tolerance principle is used, cross-allocating fragment replicas to sites containing other fragments, as shown in Figure 10.2a. However, tests have shown that the risk of network partitioning is non-negligible in the links off the main inter-city connections. This can make the replica pairs diverge.

Q.2. 2 *What can be done to address partitioning of the database replicas?*

If fragments are replicated at least in triplets, the primary-partition consistency criterion can be used, preventing divergence by allowing progress only in a partition with the majority of replicas.

Q.2. 3 *Considering that the DBMS used allowed modular fragmentation and replication, will there be situations where it makes sense to allocate different redundancy levels to different fragments?*

In the course of this observation, it was also decided to group crucial data in a main fragment (MF) and replicate it in the main site and all islands, increasing the level of redundancy of this data. This configuration study will later be extended to other data.

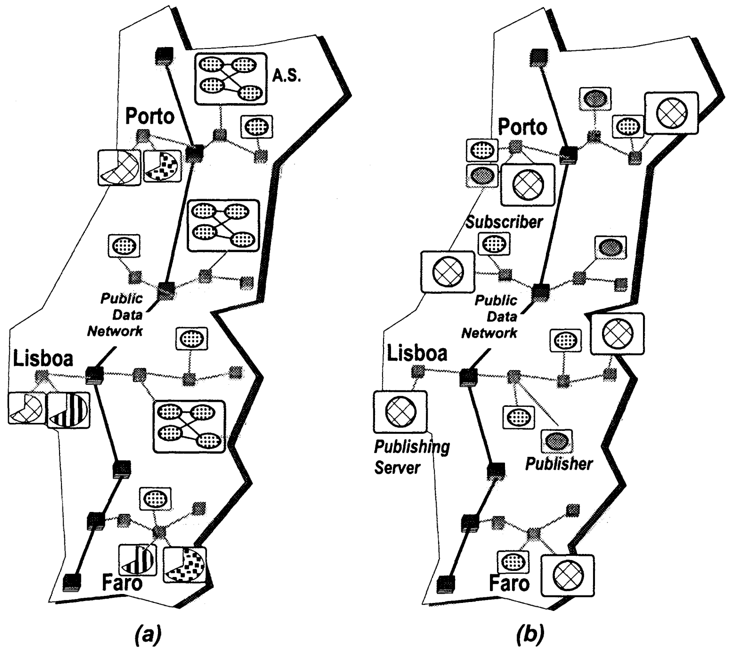


Figure 10.2. Fault Tolerance: (a) Client-Server Database; (b) Publisher-Subscriber

10.3 FAULT-TOLERANT DATA DISSEMINATION

Q.2. 4 *What can be done to improve the availability of the pub-sub server?*

With a simplex server, the initial situation depicted in Figure 5.2b, the whole dissemination system stops when the server is down. Plus, data may be lost, unless there is a transactional interface between publishers and server, and the latter has persistent storage.

A possible solution is to set up a replicated publisher-subscriber server. This may also have the benefit of improving the performance of the scheme since subscribers get in general nearer the information bus materialized by the publishing server. Figure 10.2b shows the architecture: data are published through all replicas, subscribers get data from one of them. In fact, the load of dissemination to the subscribers can be distributed by the publishing servers.

10.4 FAULT TOLERANCE OF LOCAL SERVERS

The efforts described so far were aimed at achieving fault tolerance of the global services, taking advantage of replication at other facilities. This entails two consequences: (a) undesirable service degradation as seen locally (e.g., when clients in an island have to fetch from a remote database copy due to failure of the local server); (b) unacceptable local execution glitches during reconfiguration (e.g., in a production process controller).

Two instances of the problem were identified: the main publishing/subscribing server in Lisboa, and the production control and management server in the main wine processing facility. The idea is to increase the reliability figures for the related servers, that is, taking steps to prevent failure of these components. The team has considered the utilization of either or combinations of highly-available units and clustered computers. Highly-available units are classic approaches, with embedded multiprocessors, redundant power supplies and RAID disks. Clustered computers over a fast LAN offer the advantage of modularity and use of COTS components, being in turn more complex to manage.

Further Issues

These issues need some refinement now, and the reader was assigned the study of a few questions that were still left to be solved:

Q.2. 5 *Study concrete measures based on highly-available or clustered computers to achieve "non-stop" operation of the Lisboa pub/sub server.*

Q.2. 6 *Consider an island DB fragment: study a scheme for doing local replication of that fragment, for local availability in case of one failure.*

Q.2. 7 *Study a quorum replication scheme that ensures the continued operation of the main DB fragment when both Porto and Lisboa are up and connected.*

Q.2. 8 *Study a replication scheme guaranteeing that a copy of the whole database exists in Porto. The extra fragment copies: can be read-only; need not be updated as timely as the others.*

Q.2. 9 *Define and justify the necessary communication semantics (e.g. ordering, reliability) between: publishers and publishing server replicas; server and subscriber groups.*

III Real-Time

I want a real-time system where I can log on.

— Alan Burns

Contents

Overview

Part III, Real-Time, discusses how to ensure that systems are timely, under a number of circumstances, including faults, overload, uncertainty. It is especially concerned with real-time in distributed systems. Chapters 11 and 12, Fundamental Concepts of Real-Time and Paradigms for Real-Time, address the fundamental notions and misconceptions about real-time, in a distributed context. The main paradigms are presented, in a comparative manner when applicable, such as synchronism versus asynchronism, or event- versus time-triggered operation. Chapter 12 further addresses issues such as: real-time networks, real-time processing, real-time communication, clock synchronization, and input-output. Chapters 13 and 14, Models of Distributed Real-Time Computing and Real-Time Systems and Platforms, show how to achieve timeliness of distributed systems in the several real-time classes— hard, soft or mission-critical— and models— time-triggered and event-triggered. Chapter 14 gives examples of distributed real-time systems in several settings. Chapter 15 continues the case study, this time about making the VP'63 System timely.

11 REAL-TIME SYSTEMS FOUNDATIONS

This chapter addresses the fundamental concepts concerning real-time, starting with the definition real-time and clarifying a few current misconceptions. It traces the evolution of real-time computing towards distribution and discusses its relation with fault-tolerance. Finally, the most relevant architectural approaches to real-time in networks and distributed systems are introduced, to be detailed in the subsequent chapters of this part.

11.1 A DEFINITION OF REAL-TIME

Intuitively, real-time systems are systems that deal with time. This probably means "doing things on time". But what is time anyway? And... which time exactly? The systems' time? The users' watch time? In fact, one of the fundamental problems in real-time systems design is how to relate the several dimensions of time in a distributed system: time amongst the several sites; time between a site and its users (human and other); time between the system and the environment; time between a controller and the controlled system. When thinking about how to address these problems, we face important issues concerning real-time and distributed systems:

- modeling the interaction between the computer and the real world (an oven is not a computer);
- maintaining temporal accuracy of measurements (the temperature of the oven 30 minutes ago may be useless now);

- accommodating important load versus finite resources (a 10Mbaud real-time network is useless for 20Mbaud loads);
- recognizing deadlines and urgency (an urgent message to be delivered within the next 10 milliseconds should not stay at the tail of a long queue of non-urgent messages);
- tolerating faults (no deadlines were ever met by a crashed RT computer).

When the role of time is misunderstood, a few misconceptions may arise about what a real-time system is. We will address and clarify the most common ones, after attempting at defining a real-time system.

For the treatment of the Real-Time Part, it is assumed that the reader is familiar with most of the distributed systems paradigms discussed in Chapter 2, and definitely with the material developed in Sections 2.5, 2.6, and 2.7 of that chapter. We will be using the acronym RT to mean 'real-time' in this part.

11.1.1 What is a Real-Time System?

Let us try and answer this question in a precise way. What dictates the 'real-time problem' is the need to synchronize our actions with the environment. The environment has its own pace, and thus we have to make our system adapt to this pace, and react according to the evolution of the environment. This proves more difficult than it looks, and justifies *real-time* as a research area of its own right. By convention, **real-time** (with slash) is the keyword chosen to designate this area. Do not confuse with the abstraction **real time**, the (unobservable) universal time reference, the same everywhere in the system, also called newtonian time, that marks the passage of time (*see Times and Clocks* in Chapter 2). We find as examples of real-time systems: oven controller; manufacturing cell; fly-by-wire controller; traffic lights control system; air traffic control system; multimedia computer game system; command, control and communication systems.

Real-time System - system whose progression is specified in terms of *timeliness* requirements dictated by the environment

This generic definition implies several corollaries that have been used as alternative definitions of real-time system. For example, a real-time system is a system where *the correctness of a computation is defined both in terms of the logical results and the time at which they are provided.* On the other hand, timeliness requires synchrony (*see* Section 2.6): a real-time system can be seen as a system that *has the capacity of executing actions within pre-specified intervals.* In fact, real-time can be seen as the body of principles and techniques for specifying and building *synchronous* systems. As a final corollary, a real-time system *is a system that provides at least one real-time service.* That is, although RT and non-RT services may coexist in the same system, one single RT service is all it takes for requiring the system to have real-time behavior. Examples of real-time services are: to read a sensor cyclically; to activate a valve at a precise instant; to reply to a request or deliver a message in bounded time; or to execute a task within a given interval.

There are several **classes** of real-time systems, because the kind of constraints put by matching the environment to the timeliness requirements vary. The existence of the classes is justified by the fact that different architectures and paradigms address different problems in each class, and it is difficult if not impossible, to address all of them with a single architecture. In consequence, real-time system architects are used to distinguishing between:

- *hard* real-time systems, where timing failures are to be avoided — example: on-board flight control system (*fly-by-wire*);
- *soft* real-time systems, where occasional timing failures are accepted — example: on-line flight reservation system;
- *mission-critical* real-time systems, where timing failures should be avoided and occasional failures are handled as exceptional events — example: air-traffic control system.

How is timeliness specified in real-time systems? We learned that timeliness is essentially about bounding delays. In this part, we are going to see that this may have several facets. Namely, the real-time systems community often uses terms with a specific meaning: deadline, liveline, targetline, release time, slack, laxity, jitter, latency, delay, etc. Whenever it is the case, we will try to establish the mapping onto corresponding terms in distributed systems.

A **distributed real-time system** is a particular instance of real-time system, with two major characteristics:

- timeliness guarantees have to be provided over a system of sites interconnected by a network— which is harder to do;
- once provided, timeliness guarantees can be associated to other attributes, such as modularity, geographical separation, failure independence, load balancing, and others discussed in Chapter 1— which is an additional advantage.

Some issues concerning the architecture and functionality of distributed systems assume new dimensions under a real-time perspective, which we address throughout this part:

- time-related aspects of fundamental concepts and paradigms (e.g., synchrony, time and order across multiple sites);
- models enforcing timeliness (e.g., time-triggered or event-triggered);
- scheduling (e.g., distributed and dynamic);
- communications (e.g., real-time networks and protocols);
- input-output (e.g., multiple sensor and actuator handling);
- dependability (e.g., replicated sensors, actuators and controllers).

A class of systems intimately related with real-time are the **embedded systems**, also called application-specific systems, where individual components have specific tasks or assignments. It is essentially a black-box system, made on purpose for an application. Embedded systems can be specially made from scratch, or they can be integrated with modular components such as OEM board families, or even from COTS[1] components such as PC CPU boards and peripherals. Although some authors consider them so, embedded systems are not always real-time systems. Nevertheless, they are real-time or at least interactive for their great majority, and as such a great deal of what is written about embedded systems concerns real-time.

11.1.2 Misconceptions about Real-Time

Despite the evolution and visibility of real-time (RT) computing over the past few years, a number of misconceptions about real-time still persist. There is a very complete compilation of them in (Stankovic, 1988), of which we extract the main examples:

- RT is ad-hoc design, assembly programming, interrupts, and so forth;
- RT systems are automata, pre-programmed and static;
- RT is about having enough speed, and ever-increasing MIPs and Mbauds will solve all "performance" problems;
- RT deadlines do not make sense, since they will be missed because failures occur, messages get lost, software has bugs, etc.

Real-time is currently much more than a collection of clever hardware and firmware engineering principles. If final system enhancements may benefit from ad-hoc tuning, the fact is that real-time systems design obeys a systematic that goes from architecture to programming languages, and algorithms.

The realm of static RT systems represented by PLCs, PID controllers and similar devices is but one of the facets of today's real-time systems. The "real-time systems where one can log on", many of them distributed, have pervaded the scene of interactive systems, simply because user demand has required more predictable systems in the time domain.

Many people still equate 'real-time' with 'performance'. However, real-time *is not* about performance, but about **predictability**. It is about "within 100 seconds" and not about "as fast as possible". If all actions are executed within 99 seconds each by system Slow, it performs excellently, though on average more than 100 times slower than system Fast, which executes most actions below the second. But we had not asked for that had we? Furthermore, speed will not solve all problems. The more resources we have, the more we will spend. The Wintel[2] saga is the best example available today. The point is about **scheduling** resources correctly so that *deadlines* are met. If system Fast executes one thousand actions below one second, and then takes 101 seconds to execute just one action, it will have failed.

Finally, a word about reliability: all systems fail, so the fact that measures are taken to secure timeliness specifications in normal operation does not

[1]Commercial Off The Shelf.

[2]Acronym to denote the folkloric notion that the more powerful Intel PCs become, the more power-hungry Microsoft software gets.

exempt the designer from either designing for the worst-case situation, or endowing the system with the necessary mechanisms to tolerate faults when they happen.

11.1.3 Evolution of Real-Time Computing

A few of the major milestones in the evolution of real-time systems in the past few years are enumerated in Table 11.1. Real-time in scientific terms probably started with the exploratory transatlantic navigations in the fifteenth century, with the combined use of existing knowledge on astronomy, cartography, and mechanics. Determining coarse points in the sun's movement, such as noon, evolved to measuring the minute accurately, around 400 years ago, followed later by the second, with the appearance and evolution of the chronograph (Boorstin, 1983). More recently atomic clocks, e.g. in the GPS NavStar navigation system (Parkinson and Gilbert, 1983), yield clock accuracies of the 10^{-7}th of a second.

Real-time in the realm of electronics started with control systems, made of hard-wired relay or digital systems. Then, early real-time computing systems appeared, in the form of dedicated computers for fixed-base applications, most of them military, such as SAGE or Whirlwind (Redmond and Smith, 1980). Operating systems evolved in order to provide support for development of generic real-time applications.

The advent of microprocessors opened the way for small and cheap embedded control units, modularly built around families of standard form-factor cards, relying on firmware-based real-time multitasking kernels (*see* Section 14.1). Microprocessors also gave a push to the development of black boxes such as PLCs (programmable logic controllers) and PID (proportional, integral, differential) process controllers, aimed at replacing relay and analog electronics with more versatile (programmable and settable) modules. The PLC normally polls the sensors and issues commands to the actuators based on some control program.

Distribution appeared in the form of interconnection of the above-mentioned components over real-time LANs, such as Token-bus (Token Bus, 1985), or the so-called *field buses*, simplified LANs which are a kind of digital system over a long wire, such as (CAN, 1993). MAP, the Manufacturing Automation Protocol (MAP, 1985), despite its shortcomings, was a major cultural breakthrough, bringing real-time networking into the *shop-floor*. In its first steps, distribution in computerized control was mostly concerned with replacing point-to-point cabling, through field buses: the central unit executed an automaton which read information from and sent commands to remote units on a polled, synchronized basis. More recently, we have witnessed an evolution towards using field buses as support for distributed control systems, supported by paradigms such as client-server, state-machine, or producer-consumer.

Distribution penetrated in the real-time arena for several reasons:

- geographical separation— the nature of most real-time control problems is distributed;

Table 11.1. Major Milestones in Real-Time Computing

1947	Early RT systems (Whirlwind) (Redmond and Smith, 1980)
1957	Early RT systems (SAGE) (Redmond and Smith, 1980)
1973	Rate Monotonic Scheduling (Liu and Layland, 1973)
1982	Early COTS multitasking executives (VRTX,RMX)
1983	NavStar GPS (Parkinson and Gilbert, 1983)
1983	Ada Programming Language (ADA 83, 1983)
1984	Basic Imprecision of clock synch. (Lundelius and Lynch, 1984b)
1985	Early RT LANs (Token Bus, 1985)
1985	Manufacturing Automation Protocol (MAP, 1985)
1987	Priority Inheritance (Cornhill et al., 1987)
1987	Real-Time Databases (Son, 1987)
1988	Early Field Buses (MIL-STD-1553B, 1988)
1990	Early Field Buses (FIP, 1990)
1989	Round-trip Clock Synchronization (Cristian, 1989)
1989	Network Time Protocol (Mills, RFC1119)
1991	Non-centralized Field Buses (Profibus, 1991)
1993	Non-centralized Field Buses (CAN, 1993)
1993	Deterministic (DCR) Ethernet (Le Lann and Rivière, 1993)

- decentralization— many of these problems involve interacting clusters with local autonomy;
- parallelism, load balancing, replication— desirable system characteristics come as artifacts of distribution.

Distributed real-time systems evolved along three main axes. *Embedded control and field-bus* distributed systems are mainly devoted to computerized control, and essentially try to use distributed systems techniques in low-level networks and simplified nodes. *Large-scale mission-critical* systems resort to dynamic techniques in order to secure deadlines under the uncertain circumstances encountered in the complex environments they normally address. *Soft real-time* systems represent today a great part of the interactive systems, namely multimedia rendering and conferencing systems on the Internet, and try to ensure timeliness specifications in a probabilistic way, namely through QoS negotiation and adaptation techniques.

11.1.4 Real-Time and Fault Tolerance

It is hard to think seriously about real-time without considering fault tolerance. We defined real-time system as one whose progression is specified in terms of timeliness requirements dictated by the environment, because the real world does not wait. However, it does not wait either in normal situations or in faulty situations. In consequence, ensuring timeliness in fault-free situations is complementary to ensuring it under faulty situations.

For example, suppose a worst-case message delivery time analysis for a time-critical LAN that only takes into account medium access schedulability, that is, how frame transmissions are distributed in time among the several nodes.

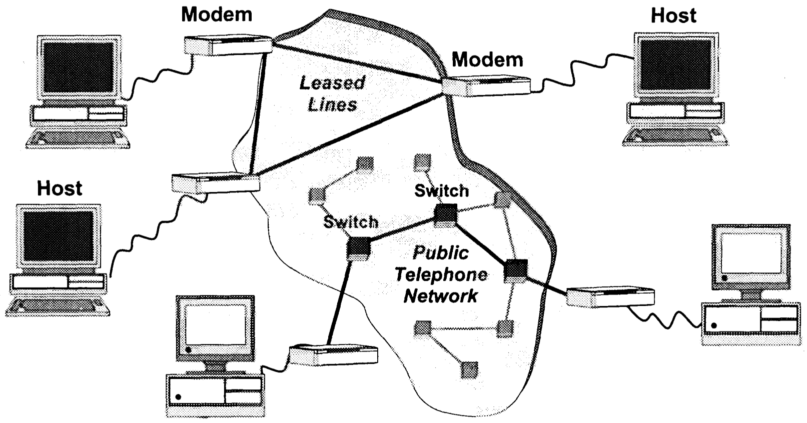


Figure 11.1. Physical Circuit

If omission faults are not taken into account, then the calculated bound will be exceeded when faults occur. If those faults are not tolerated, it will even be impossible to calculate the real bound. Even under a soft real-time perspective, we need availability to fulfill probabilistic deadline assurances. "As soon as the computer is up again" may not be compatible with the specification "any transaction should terminate within 10 seconds for at least 90% of the times, and within 1 minute in 100% of the times" with which we exemplified soft real-time systems requirements earlier. In dependability terminology, *reliable real-time* means *tolerating or preventing timing faults*, according to a pre-defined *fault model.* During this part, we will have the opportunity to see a few examples of the symbiosis between real-time and fault tolerance. It is thus expected that the reader has been exposed to relevant materials in the Fault Tolerance Part, for example, Chapter 6 and the first part of Chapter 8.

11.2 REAL-TIME NETWORKS

The first step towards setting up a distributed real-time architecture is providing the system with adequate networking structure, capable of exhibiting real-time behavior. Several network architectures serve this purpose.

A primordial structure with real-time properties is the **physical circuit** type of network. Real-time behavior, as depicted in Figure 11.1, is achieved by guaranteeing a physical path between any two endpoints, normally a combination of wires, repeaters, and switches. For example, the capabilities of telecommunications circuits for carrying real-time data are often forgotten : permanent leased lines or switched circuits such as digital telephone, ISDN or GSM. This avoids the sharing problems, such that the latency is virtually constant and equal to the propagation delay between the two points. A complementary problem is ensuring that the individual load does not exceed the throughput of the link. Physical circuits have been used to structure early dis-

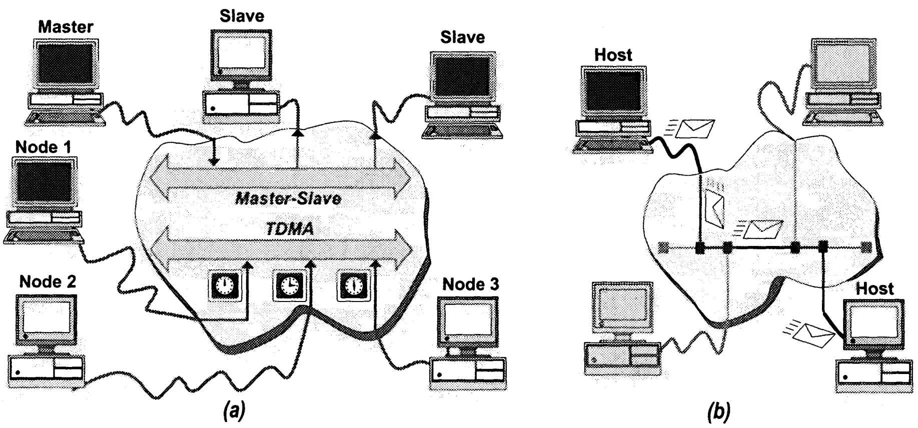


Figure 11.2. (a) Digital Bus; (b) Virtual Circuit

tributed real-time systems, namely in the fault-tolerant area (*see* Section 6.4), often resorting to special-purpose networks such as graphs of point-to-point links, or single-sender broadcast links. On the other hand, there are examples of remote control and interactive telemedicine systems built over ISDN-based computer telephony.

Another structure with real-time properties is what we might call **digital bus** type of network. Its philosophy was inherited from the centralized digital systems era, when the need arose to extend the geographical reach of control systems. Typical instantiations of the digital bus network structure are early instrumentation or field buses based on the master-slave principle, such as GPIB (IEEE-488.2), MIL-STD (MIL-STD-1553B, 1988), or FIP (FIP, 1990). TDMA-based structures are also examples of digital-bus networks (Kopetz and Grunsteidl, 1993). As suggested in Figure 11.2a, real-time behavior is ensured because although the medium is shared there is no uncertainty-causing contention, since it is shared on a pre-determined basis: a master polls all slaves; or a global clock determines the transmission slot for every node. Again, it is necessary to ensure that the global load does not exceed the throughput of the medium.

In the past few years, a number of real-time LANs have appeared, some of them standardized such as Token-bus (Token Bus, 1985), others proprietary such as DCR-Ethernet (Le Lann and Rivière, 1993). Field buses have also evolved toward a decentralized nature, emulating existing fully-fledged LANs, such as Profibus (Profibus, 1991), or CAN (CAN, 1993). These networks exhibit ring or bus shared media topologies, decentralized clocking and control, and prefigure the most common real-time network structure today, the **virtual circuit**, of the frame switching type. Real-time behavior must be achieved through a sharing policy, implemented by *medium access control* mechanisms, which ensure that a virtual path is established for each packet transmission, as suggested in Figure 11.2b. Unlike non real-time LANs, which privilege fairness of medium sharing, these mechanisms aim at letting through the most important (with higher priority) frames ahead of the others, controlling the maximum amount of time each node transmits, and/or bounding the interval between successive transmission opportunities for each node. These mechanisms aim at controlling the **latency** observed by each frame. Complementary to this, it is always necessary to match the total load offered by all nodes to the maximum **throughput** achievable by the medium.

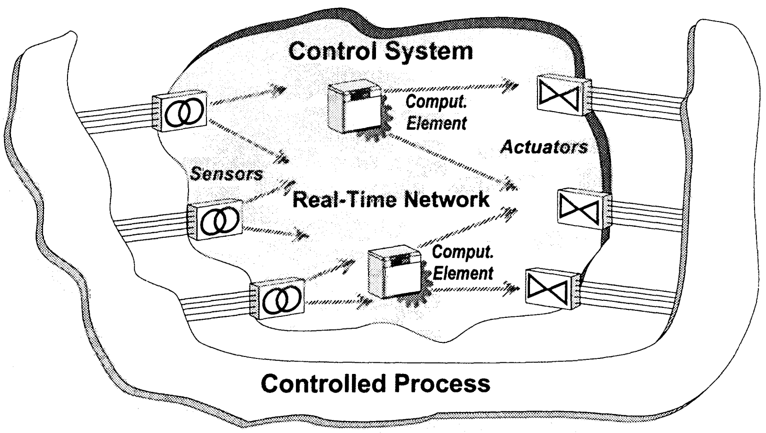


Figure 11.3. Real-Time Control Architecture

11.3 DISTRIBUTED REAL-TIME ARCHITECTURES

This section gives an overview of the main architectures for distributed real-time systems. One such architecture is the one supporting **real-time control** (real-time started with control systems). The main components of the architecture, as depicted in Figure 11.3, are the controlling system, which is the computational part, and the controlled system, the physical system under control, which we will call *environment*, for simplicity. It is very important to understand the behavior of the controlled system, since several communication and interaction paths go through it, as shown in the picture. These are called *feedback* paths. The points of contact between the controlling system and the environment are the sensors and the actuators. The control activity takes place by having the *sensors* acquire the state of the environment (e.g., the temperature of an oven, or the flow in a pipe). That information is sent to the *computing elements* of the distributed control system where it is processed, and then the reply is issued in the form of commands to the *actuators* (e.g., open the burner throttle to increase the temperature, close a valve to decrease the flow). The environment reacts to these stimuli and provides feedback as if it sent 'messages' to the sensors, closing the loop we see in the figure. Timeliness requirements are expressed differently, depending on the kind of control. Discrete control requires individual actions to be taken in bounded time, in response to events from the environment or from the controlling system (e.g., taking a robot's arm from under a 1 ton press before it comes down). Continuous control requires that a variable is maintained within an allowed value interval, although this translates, other aspects solved (such as sensing errors), to timeliness requirements: correcting it periodically, often enough to compensate for slow steady state variations; correcting it sporadically, quickly enough after disturbances (e.g., maintaining the temperature of an oven, both in steady state and after doors are opened and new, cold materials inserted). Most hard real-time systems are devoted to control and thus follow this architecture, where the main timeliness issue is to ensure that all the control loops are served frequently enough and/or sporadics are treated quickly enough.

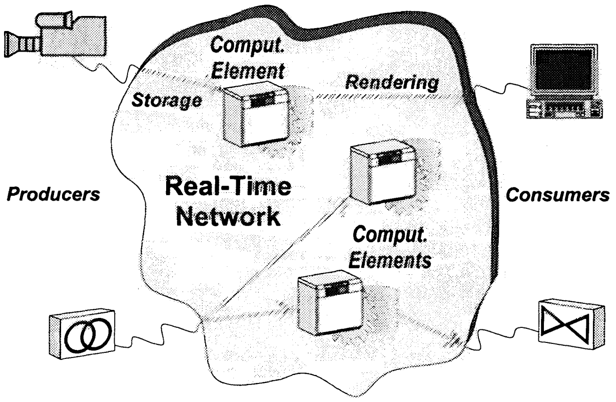


Figure 11.4. Real-Time Producer-Consumer Architecture

Another relevant architecture is the one supporting **real-time producer-consumer**. It is concerned with supplying information (data, messages) from one or more producers to one or more consumers, such that the throughput, or the latency, or both, are kept within pre-specified bounds. As depicted in Figure 11.4, data is originated in the *producers* (e.g., images from a digital camera), then processed by the distributed *computing elements* of the architecture (e.g., compressed and encoded, and/or stored), and finally delivered to the *consumers*, which absorb it. By absorption, we mean that there is no special timeliness concern associated with the information once arrived at an end consumer, that is, we might consider the consumer to be an infinite sink. Furthermore, note that in this example, data not only has to arrive at an average minimum rate (throughput), but also at a steady pace, which implies an upper bound for the instantaneous delivery latency of each message. Competing producers sending to the same set of processing nodes further complicate the issue, since it is necessary to ensure the throughput and latency properties for the whole flow. Multiple consumers just require the ability of the system to multicast the information to them in real-time. Observe that for certain applications, such as video-on-demand, the producer-consumer path is only relevant from the storage server to the consumer client, since the contents have been generated and recorded beforehand. There are hard real-time applications constructed on this kind of architecture, which normally imply totally deterministic flows from producer to consumer, such as in embedded multimedia processing systems. There are also soft real-time instantiations of the producer-consumer architecture, where constraints can be relaxed, such as Internet sound and video rendering.

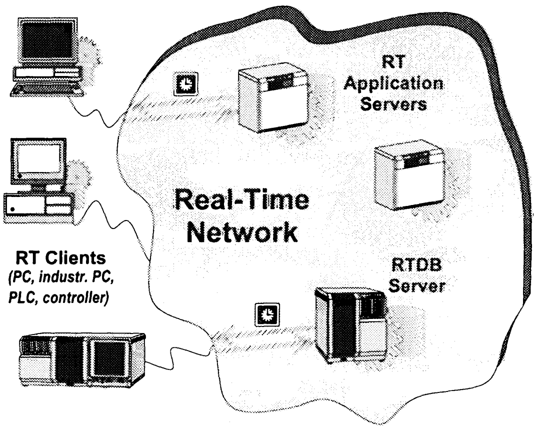


Figure 11.5. Real-Time Client-Server Architecture

Finally we analyze the **real-time client-server** architecture. Several real-time applications require the ability to execute remote commands or services and eventually return results, all of this in bounded time. A classical example of operations on this kind of architecture are transactions on *real-time database*. As shown in Figure 11.5, clients issue competitive requests to the server (e.g., dial-time request of the translation from a free-toll number to the actual subscriber number). Both the scheduling of the request for execution and the execution itself should take place in bounded time (e.g., the successful telecommunications database lookup). The result (e.g., the called party number) is returned to the client. Soft real-time applications are readily built on a client-server architecture. The more demanding hard real-time or mission-critical classes may face difficulties with traditional client-server technologies: to the lack of determinism of the arrival pattern of competing client requests, one has to add the potential lack of determinism of most multi-threaded programming approaches on the server side (*see* Section 3.6 on the basic characteristics of the client-server model). As such, real-time client-server architectures (and namely RT databases) are normally designed in a way that constrains these sources of non-determinism.

11.4 SUMMARY AND FURTHER READING

This introductory chapter discussed the fundamental concepts concerning real-time in distributed systems. Concepts and terms were introduced, such as:

real-time system; hard, soft and mission-critical real-time. The main real-time network and distributed system architectures were introduced, to be further debated in the following chapters. Namely, we divided real-time networks into three large type groups: the physical-circuit, digital-bus, and virtual-circuit groups. Finally, we discussed the real-time control, producer-consumer, and client-server real-time architectures. Relevant surveys on research in the area can be found in (IEEE-RT, 1994).

12 PARADIGMS FOR REAL-TIME

This chapter discusses the main paradigms concerning real-time in distributed systems, in the viewpoint of the system architect. Namely, the chapter addresses: specifications for describing timeliness; timing failure detection; the real-time entity-representative relation; the time-value duality of real-time entities; real-time communication; flow control; scheduling; clock synchronization; input-output. We explain these paradigms in practical terms, giving examples of the problems they solve and of their limitations.

12.1 TEMPORAL SPECIFICATIONS

The representation of temporal specifications is of extreme importance in real-time systems. Several things have to be described, specified, or quantified, when dealing with real-time paradigms: the timing of events related with the execution of real-time communication and computation actions (e.g., deadlines); the patterns of arrival of events (e.g., sporadic); the definition of triggering conditions (e.g., time lattices). We assume the reader to be familiar with basic notions about time and synchrony (*see Time and Clocks* and *Synchrony* in Chapter 2).

12.1.1 Timing of Events

Real-time researchers and developers name several timing variables in particular ways that sometimes depend on the context. Before learning these specific terms, let us analyze what we need to specify and to measure, *in general terms*, in real-time systems. Real-time systems are in essence *reactive* or *responsive*. Most of what they do is related with responding to events produced by the environment and by human users.

Response Time - the interval that mediates between the occurrence of an input event and the occurrence of the first related output event

For example, the interval between the arrival of a `train-passing` sensor reading and the output of the `close-gates` actuator command. The maximum and the minimum response times are relevant variables of a real-time system. The first, because it measures the capability of handling the dynamics of controlled processes: slow systems cannot control fast changing processes adequately. The second, because together with the first, it measures the variance of the speed of the computer response: quality of control is affected by the latter (Kopetz, 1997).

Real-time systems must respond according to pre-specified timings. The basic thing about timing is being able to specify action *durations* and event *positions* or *timestamps* in the timeline. For example: *within* T_A *from* t_A to mean that an action will be performed with a maximum duration of T_A, starting at t_A. In general terms, let us call them specifications of timed actions.

Timed Action - the execution of some operation, such that its termination event should take place *within* an interval T_A *from* a reference real time instant t_A

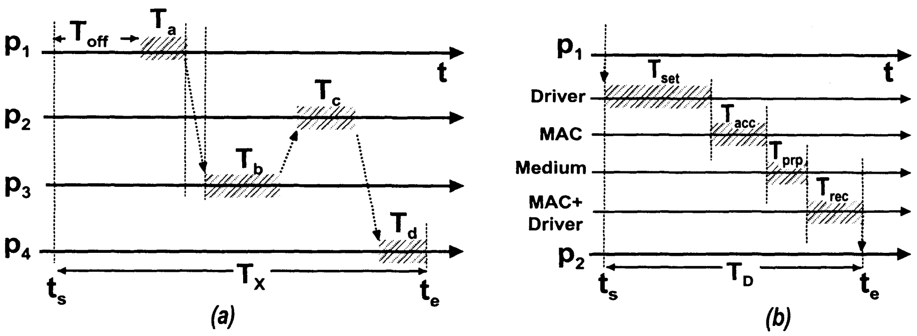


Figure 12.1. Termination Time: (a) Computations; (b) Communication

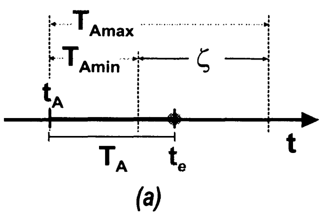
We start by analyzing the *termination time*, that is, the duration of completion of an action. Figure 12.1a shows the execution of computations. Note that the execution of an action may be complex, as depicted in the diagram of the figure: made of several elements or *tasks* that take place in several sites. Tasks are invoked for execution and run when scheduled. The essential timing specifications are: *deferral time* (T_{off}), the delay introduced before the execution is requested, also called *offset*; *termination time* (T_X), the difference between the termination (or end) and request (or start) event timestamps (resp. t_e and

t_s) in the timeline; *execution time* (T_{ET}), which accounts for the duration of the computation in continuous execution (may be shorter than the termination time), in this case the sum of durations T_a, T_b, T_c, and T_d. Figure 12.1b on the other hand shows the execution of a communication action. The figure depicts the split into the four parts that account for the termination time of a transmission: the *set-up* time (T_{set}) is spent preparing the frame for transmission, from the time it is handed from the user buffer to the operating system; *access* time (T_{acc}) is the time the frame spends waiting to be transmitted; the time spent by the frame in transit is the *propagation* time (T_{prp}); finally, the *reception* time (T_{rec}) is the time spent in transferring the frame to the recipient buffer. The delivery time T_D is the difference $t_e - t_s$ in the timeline, and is computed from the sum of the components described above.

Let us consider now how to position the *instant of completion* of an action on a desired point of the timeline, say t_A. In the case of those actions that consist of nothing else but generating an output event, this may be specified by scheduling the execution of a short-lived, negligible duration action *at* t_A, with the help of a clock. In order to achieve the same for an action that has a non-negligible and constant duration T_A, it must start at $(t_A - T_A)$, which we specify by requiring that the action terminates *within* $-T_A$ *from* t_A.

There are natural timing *errors* in the execution of timed actions. These errors are normally called *jitter* in real-time lingo, and derive from fundamental limitations of the support infrastructure, such as non-determinism of the software, scheduling, faults, clocks, etc.

Jitter - the uncertainty about the instant of completion of a timed action, taking the form of *variance* in the duration of its execution or of *imprecision* in the positioning of its termination event



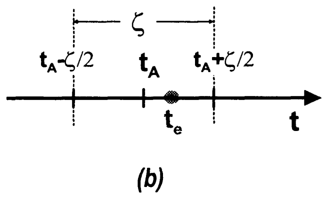


Figure 12.2. Jitter: (a) Variance; (b) Imprecision

Consider the duration specification that an action terminates *exactly after* T_A *from* t_A, that is, assuming a constant delay T_A (e.g. if it takes exactly 2 seconds for a labeling machine head to come down right over an arriving part, the former may be commanded to start coming down 2 seconds before the part arrives, for speed improvement). Assume that the execution of the action has some jitter, which shows up in this case as a *variance* in the termination time (e.g., the head may arrive too late or too soon sometimes, causing incorrect labeling) depicted in Figure 12.2a. The correct specification to recognize this fact should be *"t_e within ζ from $t_A + T_{Amin}$"*, where ζ is the action jitter, such that $T_{Amax} = T_{Amin} + \zeta$. Then, either the labeling system accommodates the jitter, or another solution must be found.

Consider now the positioning specification of an action completion event e_A in the timeline *"e_A at t_A"*. As exemplified by Figure 12.2b, jitter shows-up as an *imprecision* in the positioning of the event. In fact, in order to be correctly equated, the specification should read *"e_A within $\pm\frac{\zeta}{2}$ from t_A"*, where ζ is the action jitter. Then, either this imprecision is supposedly small enough to be acceptable, or else another solution must be found.

12.1.2 Triggering Timed Actions

There are essentially two ways of triggering timed actions in real-time systems. The **event-triggered** approach makes the system react upon the occurrence of an input event. As Figure 12.3a exemplifies, reaction of the system is immediately triggered by the event arrival, and the subsequent response issued. The system reacts as the input is made, and with the timing given by the speed of response. In the **time-triggered** approach, depicted in Figure 12.3b, the system reacts upon the command of a clock. Regardless of when an input event arrives, it is processed at the next input point dictated by the clock. Likewise, outputs are also synchronized by the clock, as shown in the figure, no matter how fast they are produced. Timed actions of different kinds can be combined. For example, input events served in an event-triggered manner, but outputs synchronized by a clock. Note that in consequence, the output jitter of an event-triggered action is given by the execution time variance, whereas in a time-triggered action it is given by the imprecision of the clock.

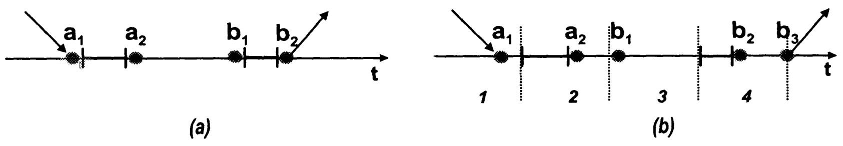


Figure 12.3. Triggering Timed Actions ($a_i \rightarrow a_{i+1}$, $b_i \rightarrow b_{i+1}$): (a) Event-triggered; (b) Time-triggered

Time lattices (*see* Figure 1.8 in Chapter 1) are powerful constructs to synchronize the triggering of simultaneous timed actions, to facilitate the measurement of their duration, and to tell the ordering of distributed events. Controlled by clocks, the same physical tick of the clock at every site marks what we call a *microtick*(Kopetz, 1997), a global tick of the lattice. Recall that events are ordered by the physical granularity (g_p) of the clock, which is often so fine that it orders events that have no causal relation (*see Temporal Order* in Chapter 2). This problem is attenuated if we construct the lattice around a global clock with an artificially coarser virtual granularity (g_v). Time is now marked in *macroticks* spaced by g_v, and all events in a g_v interval between two ticks are considered concurrent.

12.1.3 Arrival Distributions

We said that the evolution of a real-time system is dictated by the environment. This means that the system receives *inputs* from the environment, and then has to react according to what is expected, and when expected. We spent the last sections analyzing how to specify and trigger this response.

But what about the inputs themselves? Can we process an undefined amount of information per time unit? Of course not, systems have a limited processing capability. In consequence, all variables that we have analyzed, such as response time or termination time, depend on the load on the system. Can we predict and/or bound the amount of information that arrives at the system input? Only in some cases. This is why we have defined classes of real-time systems: (a) the determinism of the system with regard to time depends on the predictability of the inputs received from the environment; (b) the predictability (or determinism) of the environment depends on the class of application. Systems handling regular and deterministic arrival patterns are simpler, but have to cope with the potential lack of coverage of those assumptions. Systems accepting irregular and uncertain distributions are closer to physical reality, but are more difficult to design and prove correct.

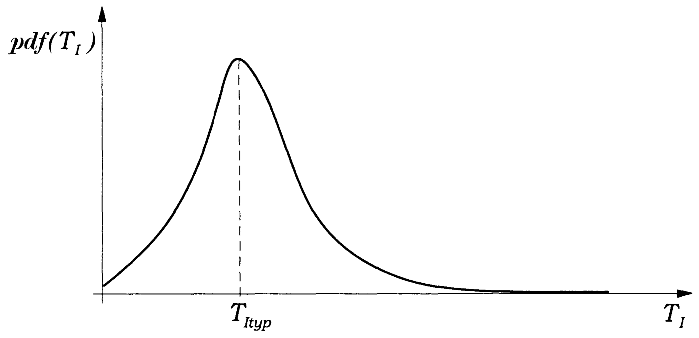


Figure 12.4. Example Probability Density Function of Event Inter-arrival Times T_I

In Figure 12.4 we depict a probability density function that could model the inter-arrival intervals of events produced repetitively by a source. Such a distribution has a zone (in the center) where events are distributed more or less every T_{Ityp}. However, there is no limit to the amount and frequency of information that may arrive to the system on certain occasions, since the distribution intersects the Y axis. We call such distributions **aperiodic**. It is not hard to see that aperiodic distributions are not ideal to achieve real-time behavior because response time is conditioned to the amount of information input per time unit, of which aperiodic distributions offer no guarantees.

On the other hand, if events arrive in a known maximum amount at known points of the timeline, i.e., in a regular fashion, it is easy for the system to behave deterministically, and for response and termination times to be precisely

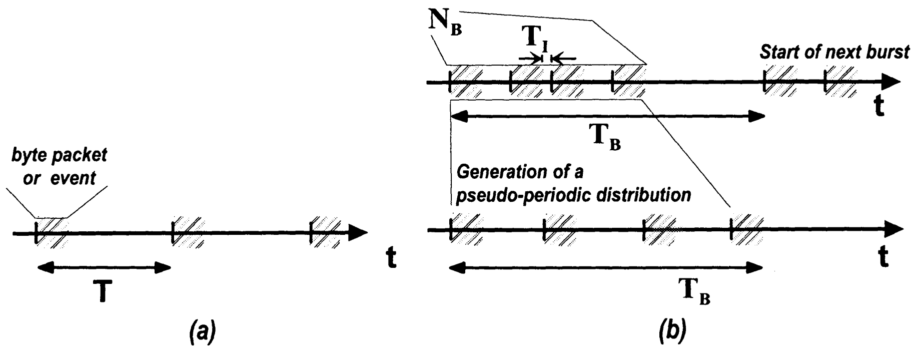


Figure 12.5. Discrete Arrival Distributions: (a) Periodic; (b) Sporadic

determined. In Figure 12.5a we depict such a distribution, with the obvious name of **periodic**, of period T.

Sometimes the designer must achieve deterministic behavior, but she cannot model the environment so regularly, because the latter is rather unpredictable, with irregular arrival patterns, such as event-triggered message deliveries and task execution requests from sensor events. A more sophisticated discrete arrival pattern represents this behavior, the **sporadic** pattern, which assumes that input information arrives irregularly in bursts, but has a few crucial bounds. Figure 12.5b (top) graphically shows the several parameters involved:

- **burst period** (T_B)– minimum delay between the start of two consecutive bursts, a known lower bound
- **burst length** (N_B)– maximum amount of information submitted in one burst (e.g., number of events, number of bytes), a known upper bound
- **inter-arrival time** (T_I)– minimum separation between two consecutive events, a known lower bound

A sporadic arrival distribution is normally treated under a short-term perspective: sporadic requests have to be served within the inter-arrival time (T_I), and an amount up to N_B of such requests must be handled without resource disruption. This is adequate for example for emergency events. However, a sporadic distribution is one that has long-term regularity, and as such it can be mapped onto a periodic distribution. This is adequate for treating sporadic events that are generated by periodic tasks, and also the so-called *event showers*. As a matter of fact, under these conditions N_B/T_B gives the rate of a periodical distribution where we would have spread the event arrivals throughout the period interval, as the bottom of Figure 12.5b suggests.

In conclusion, the real-time system architect should not forget that arrival distributions are mere artifacts to represent the environment behavior. They serve to separate concerns, and prove that given a problem and an event distribution, there are paradigms solving the problem for that distribution. However, the importance of the other part of the job should not be minimized, as it is sometimes the hardest one: to prove that the environment is faithful to the distribution we chose to model it. Nothing could go more wrong than a wrong set of assumptions (*see Failure Assumptions and Coverage* in Chapter 6).

12.1.4 Utilization Factor

Another way of determining whether *time is enough* is in relative terms.

Utilization Factor - measure of the percentage of useful work time of a resource, over elapsed time

For example, an utilization factor of 70% is often pointed out as a good CPU time-loading figure for microprocessor based control systems (Laplante, 1997). It means that during a period of 100ms, the processor is executing useful instructions during 70ms. Put the other way around, it also measures whether the CPU has enough power to handle a set of tasks: if 3 tasks each with a duration of 40ms on a given CPU have to complete within 100ms, the CPU utilization factor becomes 120%, which means it is *overloaded.*

Communication resources are also measured in terms of channel utilization factor. An utilization factor of 85% on a 10Mb/s Token-bus LAN means that during the period of a second, the channel is letting 8.5 megabits through. The maximum utilization factor (10 megabit per second in this case) is also called maximum *throughput.*

12.2 TIMING FAILURE DETECTION

Note that hard real-time systems are designed in terms of preventing timing failures. However, controlled timing failures are allowed in mission critical or soft real-time systems. In consequence, the measures to be taken by real-time systems in response to timing failures vary according to the class of operation: orderly fail-safe shutdown; recovery or compensation; reconfiguration by adaptation to less stringent deadlines. Whatever the solution, timing failures should be detected. In fact, we are concerned with *late timing failures* (e.g., the missed deadline or late message syndrome) which are the most general type of omissive failure (*see Fault Assumptions and Coverage* in Section 6.1). So, more precisely, a timing failure is:

Timing Failure - Given the execution of a timed action specified to terminate until real time instant t_e, timing failure is the occurrence of the termination event at a real time instant $t_e^{'}$, $t_e < t_e^{'} \leq \infty$. The amount of delay, $Ld = t_e^{'} - t_e$, is the lateness degree

What do we need to know about a timing failure in a real-time system? We need to detect it in bounded time, i.e., in a *timely* manner. We must detect all relevant late actions as timing failures, i.e., in a *complete* manner. We must avoid detecting timely actions as failures, i.e. in an *accurate* manner. Recall that we have already discussed *crash failure* detectors (*see* Section 7.1), having characterized the quality of detection according to two essential properties, completeness and accuracy. We should be able to characterize timing failure

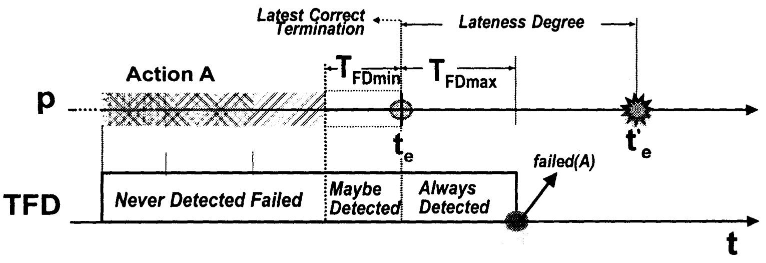


Figure 12.6. Timing Failure Detection in Action

detectors according to a similar reasoning. We must however introduce the time dimension, in order to define both the failure and when it is detected. The following defines the desirable properties of a **timing failure detector** (TFD), assuming that any timed action has an observable termination event e, specified to occur until real time instant t_e:

Timed Strong Completeness - There exists $T_{TFD_{max}}$ such that a timing failure in any timed action is detected within $T_{TFD_{max}}$ from t_e

Timed Strong Accuracy - There exists $T_{TFD_{min}}$ such that any timed action that terminates at p before $t_e - T_{TFD_{min}}$ is considered timely

The properties of the failure detector are illustrated in Figure 12.6. Note that 'timed' in both properties specifies that there is an upper bound ($T_{TFD_{max}}$) on detection latency, and a lower bound ($T_{TFD_{min}}$) on detection accuracy.

12.3 ENTITIES AND REPRESENTATIVES

There is a fundamental paradigm that has to do with the relation between elements of the environment and their computational representation, the *entity-representative* paradigm (Kopetz and Veríssimo, 1993). This relation is more important than meets the eye, and if neglected, the possibility of separation of concerns in the conception of parts of the real-time systems is diminished. That would be a bad architectural principle.

A **real-time entity** (RTe) is an element of the environment, such as a fluid valve or the temperature of an oven, with a behavior which may be time-dependent and a state the system is supposed to acquire or modify, described by continuous or discrete values, e.g.: 150 meter/sec; open/closed. The state of RTe's can be read or written to, but not both. Consider the example of a valve. We should define a write-only RTe `valve_actuator` representing the valve actuator, whose state can be positioned to one of `open|closed`, by actuator-dependent procedures (e.g., commands, control registers, etc.). Then, if we would at the same time wish to monitor the state of the valve, we should define a `valve_sensor` read-only RTe, whose state might take one of the values `open|closed`.

The **representative** (RTr) of a real-time entity is an element of the computational system through which the latter *observes* or *acts* on the state of the real-time entity in the environment. Representatives offer an interface tractable by the other elements of the computational system, and resort to sensors and actuators as a means of handling the RTe's they represent. They can assume several forms: an integrated intelligent sensor controller; the driver of a stepping motor; a computer process controlling PC I/O boards; an A/D (analog-to-digital) conversion acquisition module.

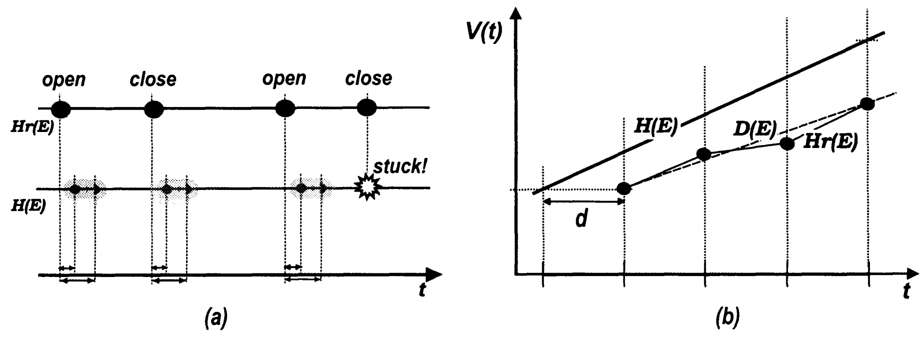


Figure 12.7. Real-Time Entity-Representative Relationship: (a) Discrete Entities; (b) Continuous Entities

Note that the state of a real-time entity is not accurately reflected in its representative at all times during system evolution: the temperature of an oven and its measure at a sensor representative differ with sensor *accuracy* and, more importantly, with *time*; a valve actuator representative may have been instructed to shut, but the valve itself may remain open because of a *failure*. The problem is amplified by ***distribution*** and *replication*, since one can get divergent readings of the same RTe by different sites. A correct design should address these problems (*see* Chapter 13).

For the sake of understanding the relation of an entity E with its representative $r(E)$, consider the evolution of the state of E with time as a non-observable sequence of states along the timeline. Then, consider the evolution of the state of $r(E)$, composed of a sequence of observations or actuations, depending on whether it is a read-only or a write-only RTe, respectively. The real-time entity-representative relation is represented in Figure 12.7, where the curves of $E(t)$ and $r(E)(t)$ are represented respectively as $\mathcal{H}(E)$ and $\mathcal{H}_r(E)$. In essence, a representative *emulates* its real-time entity with an error in the value of state, or in the time of state changes, or in both.

Figure 12.7a illustrates $\mathcal{H}(E)$ and $\mathcal{H}_r(E)$ for a write-only boolean RTe (e.g. valve). The exemplified window of discrepancy between the RTr inside the computer system ($\mathcal{H}_r(E)$) and the actual RTe ($\mathcal{H}(E)$) has a fixed part, the actuation delay, and a variable part (in gray) because of the jitter of positioning the command. In faulty situations, the discrepancy may be so large that the relation is no longer valid, unless measures to ensure fault tolerance are taken: in the last actuation, the RTr says `open`, whereas the valve got stuck at `closed`. Figure 12.7b illustrates an analog read-only RTe. Against the curve of $\mathcal{H}(E)$, we depict $\mathcal{H}_r(E)$, which exemplifies the discrepancy between the RTe and its representative. What happens may be explained by three factors: a delay in reading the state of E (execution time); the jitter of that reading (variance in execution time); and the magnitude error of the reading itself (sensor inaccuracy). Dashed line $\mathcal{D}(E)$ shows what would be the situation if only a fixed delay error component d existed.

The entity-representative relation is an important architectural paradigm. Representatives hide the complexity of the physical reality, transforming it into representations tractable by computers. For the real-time computer system, $r(E)$ *is* E. This transformation can be extremely useful, provided that the resulting errors are definable and/or bounded. The paradigm allows a separation of concerns in the design of real-time systems:

- *definition of the real-time entities* – response to the environment part of the requirements specification
- *definition of the representatives* – specification of the computational entities representing the environment;
- *definition of the input/output* – specification of the reliable and timely observation of, and actuation on, the state of RTe's;
- *definition of the control* – specification of the reliable and timely processing of the information supplied by input representatives and production of responses to output representatives.

12.4 TIME-VALUE DUALITY

Consider the specification of an action: "at real time instant T_A, produce a result of value V_A". Upon execution, the action will produce a value v_A at real time t_A, for storage, use, or other. For example, delivering a message, or storing an observation. Producing a late correct value yields a *timing error*. Likewise, producing a timely incorrect value yields a *value error* (*see* Section 6.2).

Now assume that the specification concerns the state of a real-time entity E, *whose value depends on time*, $V = E(t)$. The relevant action specification becomes: "at real time instant T_A, produce a result of value $V_A = E(T_A)$". For example, determining the position of an engine crankshaft. Producing $v_A \neq V_A$ at $t_A = T_A$ yields a *value error* $|v_A - V_A|$, on time (e.g., an erroneous position of the crankshaft at time T_A is returned). Likewise, producing $v_A = V_A$ at $t_A > T_A$ would just yield a *timing error*.

Would it? Recall that by specification the value returned at t_A must be $E(t_A)$. However, what was returned was $E(T_A)$ (e.g., the value of a past position of the crankshaft). So, in this situation, the timing error causes a value error, since the "correct" value returned concerns the value of the entity in the past. If the timing error is $|t_A - T_A|$, the corresponding value error is $|E(t_A) - E(T_A)|$.

Essentially, a **time-value entity** is such that there are actions on it whose time-domain and value-domain correctness are inter-dependent. This paradigm consolidates notions addressed in the context of a number of problems in different areas of computing, such as computer control, I/O sensing, real-time databases, or clock synchronization (Kopetz, 1997; Poledna, 1995; Marzullo, 1990; Ramamritham, 1995; Lamport and Melliar-Smith, 1985). There are two problems to solve if correct operation of a system using time-value entities is sought:

- ensuring the correct observation of both an instantaneous value of the entity and its positioning in the timeline, that is, the *time of a value*, which assumes two facets:

  observing the value at a given time – e.g., the angle 50μsecs past the lower position of the crankshaft, a switch position at 5:00

  observing the time at which a given value occurs – e.g., when is the crankshaft at 5 degrees to the top position, whenever a switch closes
- ensuring the correct use of such an observation past the time it is made, that is, the *value over time*

The first problem is equated with the error in observing the time of a value, i.e., the observation error. For an observation $\langle r(E_i)(t_i), T_i\rangle$ of the value of an RTe E_i at t_i receiving timestamp T_i, the **observation error** in the *value domain* is given by $\nu_i = |(E_i(T_i) - E_i(t_i)) + (E_i(t_i) - r(E_i)(t_i))|$. The first term in parentheses is the effect of the timing error in positioning the observation, supposedly at T_i but in fact made at t_i. The second is caused by the error of the observation apparatus (sensor module). Simplifying, we get an intuitive $\nu_i = |E_i(T_i) - r(E_i)(t_i)|$: we expect the value of E_i at T_i, but we get an approximation of the value $(r(E_i))$, measured approximately (t_i) at T_i. For observing the time at which a given value occurs, and in fact the sensible way to observe discrete entities (i.e. ones whose value jumps abruptly, say from logical one to zero), we should use a time domain metrics. The observation error in the *time domain* would then be $\zeta_i = |T_i - t_i|$: E_i assumed a given value at t_i, but the system logs it as having happened at T_i. The error accounts for the positioning error (jitter), and the time-domain effect of the sensor value error. Since determining the exact error of each observation is not feasible, we may instead work with an upper bound:

- *Given a known $\mathcal{V}_o$, we say that an observation $\langle r(E_i)(t_i), T_i\rangle$ is* **consistent** *in the* **value** *domain, if and only if $\nu_i \leq \mathcal{V}_o$*
- *Likewise, given a known $\mathcal{T}_o$, we say that the observation is* **consistent** *in the* **time** *domain, if and only if $\zeta_i \leq \mathcal{T}_o$*

The problem can be generalized to the values of a set of RTe's *at a given instant*, e.g., needed for performing computations to derive a composite variable. If the relevant observations all have bounded errors referred to that instant, the total error of the computation is bounded:

- *Given a known $\mathcal{V}_m$ and a set of observations of RTe's, we say they are* **mutually consistent** *in the value domain, if and only if there is an instant t_m such that each observation $\langle r(E_i)(t_m), T_i\rangle$ is consistent w.r.t. $\mathcal{V}_m$*

It is a sufficient condition for a set of observations to be mutually consistent, that they are consistent, and that the timestamps of all observations fall within a known interval $\mathcal{T}_m$ (i.e., $\forall i,j\ |T_i - T_j| \leq \mathcal{T}_m$). The latter is also called the *relative validity interval* in the context of databases (Ramamritham, 1995; Song and Liu, 1992), and is also related with the notion of *accuracy interval* in the context of replicated sensor observations (Marzullo, 1990).

The second problem we stated is concerned with using a value while it is still valid. In order to solve this problem, we must bound the response time or the termination time of the action that follows the observation (unexpected or programmed) of a time-value entity. The action can be performed later than the observation time, but not too late or unpredictably later, because this may imply performing an incorrect action. Assume bound $\mathcal{V}_a$ for the maximum acceptable error accumulated by an observation over time, depending on the application in view and on the dynamics of the observed time-value entity (in order to separate concerns, we neglect the observation error and define T_i as the instant of reference):

- *Given a known* $\mathcal{V}_a$*, we say that an observation* $\langle r(E_i), T_i \rangle$ *is* **temporally consistent at** $t_a \geq T_i$ *if and only if* $|E_i(t_a) - E_i(T_i)| \leq \mathcal{V}_a$

In complement to the "instantaneous" consistency property of the observation instant, this property captures the evolution of consistency with time, a characteristic of time-value entities. In fact, temporal consistency can be secured if an interval $\mathcal{T}_a$ can be defined such that the variation of the value of the RTe within that interval is at most $\mathcal{V}_a$. In other words, an observation is temporally consistent *within* $\mathcal{T}_a$ *from* T_i. This interval is also called ***absolute validity interval*** for databases (Ramamritham, 1995; Song and Liu, 1992), or *temporal accuracy interval* for control (Kopetz, 1997).

The time-value paradigm is at the heart of practically all real-time designs. It gives a common explanation to phenomena that have been addressed separately, such as temporal constraints in R/T databases, dynamics of computer control, and even clocks. In fact, clocks are interesting time-value entities: the value of a clock, $c(t)$, is a value established at a given time, which represents time itself. As an exercise, the reader may wish to find out the analogies between the properties of time-value entities and those of clock systems.

12.5 REAL-TIME COMMUNICATION

Real-time communication is related with achieving a few fundamental attributes:

- known and bounded message delivery delay
- deterministic time-domain behavior
- recognition of urgency classes in the overall traffic
- reliability of medium connectivity

Depending on the type of architecture, these are achieved in different ways, but whatever the techniques, the real-time communication paradigm can be expressed in generic terms:

Real-Time Communication - the achievement of bounded and known message delivery delays, in the presence of disturbing factors such as other real-time traffic, variable load, or faults

This generic definition suggests a few things. Firstly, that it is necessary to transmit frames (network-level information packets) in bounded time, given the interference of other traffic. Some of it will be competing for the channel on an equal foot, but clearly not all frames have the same importance, or *urgency* class, a parameter that allows some frames to get through ahead of others. Practical communication systems hardware (e.g., LANs) provides this distinction through priorities. Besides, channel scheduling should encompass the fact that load may not be steady, which means that the *latency* bounds should be achieved under variable throughput, known to be a difficult goal.

Another issue suggested by the definition is that it is necessary to get the message (user-level information packet) through despite faults. Real-time touches reliability in this point. It is of little use guaranteeing schedulability of individual frames on the network without thinking what is the final time budget to get a message across. This encompasses the use of mechanisms studied in Chapter 7, to detect, recover or mask transmission errors.

The discussion above suggests one of possible strategies for realizing the real-communication paradigm, a divide-and-conquer strategy breaking down a solution in a number of conditions to be fulfilled:

1. **computing the load budget and defining urgency classes**– defining urgency classes and allocating worst-case load patterns to each

2. **ensuring connectivity**– providing the adequate measures to ensure reliability of the network medium, and control partitioning

3. **preventing timing faults**– enforcing a bounded time from request to actual transmission of a single frame, given the worst-case load conditions assumed, in absence of faults

4. **tolerating omission faults**– ensuring that a message is delivered despite the occurrence of omissions

5. **controlling the flow of information**– ensuring that the offered load is such that the desired throughput and latency are secured

The conditions presented above are sufficient to achieve the real-time communication requirement. Condition 1 makes the basic assumptions about the environment. Condition 2 encompasses the initial discussion about how to achieve medium reliability. Both conditions have to meet the requirements of the application in mind. Condition 3 makes sure that any frame is sent within a known time bound, even if it does not arrive. Condition 4 ensures that a message is delivered in the presence of omission faults. A time bound as per Condition 4 is calculated as a function of the assumed maximum number of omissions during the protocol execution, the use or not of space redundancy, as per Condition 2, and the time bound on individual transmissions as per Condition 3, if time redundancy is used. Condition 5 addresses the need for matching the actual load presented by the environment, assumed initially by Condition 1, with the capabilities of the system. A real-time communication model implementing this strategy in presented in *Real-Time Communication Models* in Chapter 13.