2
Concepts of Concurrency

In this chapter, we cover

- Architectures in concurrency
- Naming and addressing
- Sharing and synchronisation
- Mutual exclusion, race conditions, semaphores and monitors
- Timing
- Dependability
- Types of servers
- Clusters, load-balancing and Grids

2.1 Overview

A practical computer system with concurrent elements will encounter some common patterns. There are a few *architectures* which we commonly see; as part of this, we need to *name* and *address* objects within that architecture in order to manage these objects effectively. Among these objects will be resources that are essential for the system to successfully accomplish its tasks.

Next we consider the *sharing* of data amongst concurrent objects and

synchronisation of those objects. There are some classical features from computer science that we encounter at this point: *mutual exclusion* and *semaphores*.

A particularly interesting aspect of concurrent systems involves *timing*. A system that has many simultaneous demands that must be met by *hard* deadlines is one motivation for implementing a concurrent system.

The behaviour of a system as it *fails* needs to be considered, particularly as individual elements fail. So a *dependable* computer system will need to be *available* and *reliable*: we often achieve this by means of *replication* — which itself brings complications.

We finish this chapter with some discussion of some architectural issues in implementing concurrency, particularly the different types of servers that are relevant in a distributed system, as well as related issues associated with load-balancing, such as clusters and Grids.

2.2 Architectures in concurrency

We identified, in Chapter 1, two broad types of distributed systems:

Client-server p.5

- Client-server
- Peer-to-peer

Peer-to-peer p.5

The client-server model, as we discussed earlier, is widely used, particularly in web applications and multi-player computer games. It has advantages in terms of scalability (e.g., it is usually relatively easy to add new servers to improve performance) and maintainability (e.g., the server, if well designed, can be updated without needing to update the clients).

The peer-to-peer model is growing in use, and has advantages in terms of uniformity and scalability (and in some cases, performance), though some peer-to-peer applications are more complex than client-server applications. We explore some of these differences in later chapters.

Both peer-to-peer and client-server architectures rely on a common set of techniques and methods in their construction and execution, and we now discuss some of these fundamental ideas.

2.3 Naming and addressing

In a distributed system, where a number of different objects exist that provide resources and services, these objects must be given *names* so as to identify

them. Objects may include pieces of software and hardware, but also abstract concepts such as access privileges and descriptions of how to use these objects (e.g., in the case of web services). The *owners* of objects should also have names. This is particularly important from the perspectives of security and reliability. The key characteristics of names are:

- they should be *memorable*, since they are intended to be used by programmers to find objects;
- they should *uniquely identify* an object to the programmer.

It is important to note that while a programmer should have a unique name for an object, this name may differ from that given by the network infrastructure to the object. The name by which a programmer refers to an object is usually called a *logical name*, while the name that the system uses to refer to an object is called the *physical name*. Programmers will have different goals from developers of network infrastructure, who may be more concerned with preventing access to objects except via the logical name. Moreover, physical names are often cumbersome to work with for a programmer, since they often include long strings of bits that are difficult to manipulate for anything other than a machine.

When the idea of names is introduced, we must also introduce the idea of an *address*. In its simplest form, an address provides the means to locate an object in a system. An address might be as simple as a memory location, but it may be more complex: a reference to a directory of business services, or a chain of references that, when followed, produce the object of interest.

2.3.1 Examples of names and addresses

Our first example of a name is an *email* or *mailbox*, such as john.smith@yahoo.co.uk. This is a unique logical name that can be used to locate a specific user. This name also contains information about how to locate the user: the string yahoo.co.uk. This information is used by the Internet *Domain Name System (DNS)*, which will be explained in more detail shortly.

Another important example of a name is a *Uniform Resource Locator (URL)*. These are used to uniquely identify objects on the web.

```
http://www.cs.york.ac.uk/research
```

uniquely identifies the research page via a globally understood address (`www.cs.york.ac.uk`). Note that this example is of a logical name; it describes how to find the object, not its physical location. The physical location of either the

globally understood address, or the research page, may change at any time, and the above name would still be able to find the object.

Multipart logical names, like URLs, are commonly used, particularly in hierarchical systems such as the web, or file systems. In file systems, a multipart logical name uniquely identifying a file specifies a chain of references in a hierarchical directory system, for example

```
/usr/bin/proc/id.c
```

This again specifies how to find the object `id.c` by following a logical chain of names. It does not specify a physical location for `id.c`; it simply describes how to find this object.

Another example of logical names is the addressing scheme of the *Internet Protocol (IP)*. This is (at least conceptually) what is used to identify a computer on the Internet; sometimes, confusingly, it is also called an IP address. No two computers can have the same IP address simultaneously. IP (version 4) addresses are 32 bits long, and are traditionally written as a 'dotted-quad', e.g., 141.163.7.212. IP version 6 (a new network layer protocol), designed to make more IP addresses available, uses 128-bit addresses, normally written as eight blocks of four hexadecimal digits: for example 2007:0eef:0000:0000:0000:0000:3323:553a.

BSD socket addresses p.83

A key question at this stage is: how do logical names resolve to physical names? Each object in a distributed system is physically located somewhere: on a disk, in memory, on a server, etc. It is therefore necessary to have the means to map logical names to physical names and locations, i.e., an *address mapping* facility. We explore several examples of address mapping mechanisms shortly.

One important issues with names and addresses in distributed systems is *scale*: in a realistic distributed system, involving thousands if not millions of objects, each must be uniquely identified, and sets of logical names carefully managed. Schemes for managing names, and address mapping mechanisms, must scale up if they are to be useful in a distributed setting.

2.3.2 Address mapping mechanisms

Distributed systems use a number of address mapping mechanisms to allow programmers to work with logical names instead of more cumbersome physical names. We explain, briefly, how three of these mechanisms work: *name servers*, the *Domain Name System* (which is a key technology for the Internet), and the *Universal Description, Discovery, and Integration (UDDI)* mechanism at the heart of web services systems.

2.3.2.1 Name servers. The simplest and most general addressing scheme uses *name servers*, which provide unified support for mapping the names of users, services and resources to addresses. A name server is effectively a large table of name-to-address resolutions. When a client wants to use an object in a system, they send a request to a name server, consisting of the name of the object they want to use. The name server then looks up the address of the object and returns it. Subsequent uses of this object need not go through the name server; they can use the returned address directly by caching it locally. Of course, if the name server is updated with a new address for an object, it may need to broadcast these changes to clients.

There are three points to note with name servers:

- Clients must know the address of the name server in order to make requests.
- The performance of the distributed system is tied to the name server: for this reason, a set of cooperating name servers rather than a single name server is preferred. This is also helpful for reliability.
- The reliability and security of the distributed system is tied to the name server: if it fails, or if it is compromised, no guarantees can be made about the overall reliability or security of the system as a whole.

2.3.2.2 Domains and the Domain Name System. The *Domain Name System (DNS)* is a standard distributed addressing mechanism that provides efficient name-and-address lookup for the Internet. It is a hierarchical mechanism based on the notion of *domains*.

In DNS, users are placed in individual domains; names are unique in domains, and a user address is called a *domain name*. These domain names must be registered with the DNS. The domain naming system is hierarchical and multilevel. For example, the domain cs.york.ac.uk is valid in DNS. It should be read from right to left. The top level (and most general level) of the domain is uk, which contains the domain ac.uk, which contains the domain york.ac.uk, which contains the domain cs.york.ac.uk, which is the Department of Computer Science at the University of York, an academic institution in the UK.

DNS is used for resolving names into addresses for email and file transfers (including those involved with the web, e.g., resolving web addresses). The address returned by DNS is the IP address of a host computer on the Internet; this is a unique value.

The database used by DNS is implemented using a set of name servers. Each name server holds part of the name database that logically corresponds to its local domain, as well as information about other name servers. Typically,

each domain has an authoritative local name server. These additional name servers are passed requests that cannot be resolved by the local domain server.

The DNS stores further information: it includes mail exchanger records so that Internet mail systems can find an appropriate machine to send emails. Other records include CNAME, which give aliases (additional names) to existing domain names.

2.3.2.3 Universal Description, Discovery, and Integration. *Universal Description, Discovery, and Integration (UDDI)* is a more recent mechanism for managing names. It is particular to the construction of web service systems. It provides a standard way to organise web service objects and allows programmers to find and use them. In this sense, it is much like DNS and any other name server. UDDI has the following distinctive properties:

- New objects can be added via a process called *publishing*. Objects are described in a standard way, and this standard interface is made available for other services and programmers to find and use.
- The UDDI registry is based on XML, a standard markup language that is at the heart of web service systems.
- A standard set of protocols, SOAP, is used for interacting within the UDDI. These protocols are XML-based and sit above TCP/IP.

The main difference between UDDI and general-purpose name services is the degree of precision that is needed and can be obtained with UDDI. Effectively, when a name of an object is looked up with UDDI, this object is an interface to a software service on the Internet. The client requests an object that conforms to a specific interface (i.e., provides the specific services that the client needs).

2.4 Sharing and synchronisation

A key aspect of distributed systems, particularly modern ones like web services and Grids, is the *sharing* of objects and resources. For example, an object responsible for validating credit card details with a card vendor can be shared amongst a number of companies. In order to support sharing and resolve *contention* amongst clients, objects and resources must be allocated to them, and clients must demonstrate that they are legitimately allowed to be allocated a resource. This last issue, involving *authentication* and *authorisation*, will be discussed in more detail in Chapter 7.

When dealing with resource contention, we must *allocate* limited resources

to individual users or processes, while avoiding problems such as *race conditions* (discussed in the next section).

Race conditions §2.5.1, p.18

2.4.1 Allocation of resources

Resources that can be allocated, perhaps according to a schedule, are generally of two kinds: unique resources (e.g., a file, the CPU), or a set or pool of replicated resources.

Allocation for a unique resource is often done on a *first-come first-served* basis, i.e., the first request for a resource is the first to be served, although other methods are also available. This mechanism often involves *locking* access to the resource, so that only one individual can use it, and *unlocking* the resource when the individual is finished with it. It also involves queues of requests, so that once the resource has been unlocked the next request in the queue can be processed. Mechanisms for locking resources, such as semaphores, will be discussed in the next section.

This scheme may be too conservative for all resources: consider accessing a file. Multiple individuals can safely read the file at the same time, but only one can write to the file at a time, otherwise data could be lost or corrupted. In this case, we may need to apply a *synchronisation* mechanism to support mutual exclusion. Synchronisation involves coordination of processes and resource accesses with respect to time. This is also discussed in the next section.

Allocation from a pool of resources can be done in a number of ways, including first-come first-served, or a priority scheme, where requests are given some application-dependent priority ordering. The difficulty with priority ordering is *starvation*: one request may always have a lower priority than other requests, and thus will never be allocated a resource. Mechanisms for avoiding starvation will be discussed in the next section as well.

2.4.2 Example: File synchronisation

An interesting example of a software system that supports sharing and synchronisation is a file synchronisation framework called Unison, due to Benjamin Pierce [54]. It allows two replicas of a collection of files to be stored on different computers, connected via a network and, modified separately. Differences between the replicas can be reconciled, and they can individually be brought up to date by having the modifications propagated to the different computers.

Unison Ch.9

Unison works by taking snapshots of the state of the world (i.e., the

replicas being synchronised, as well as its internal state) at every point in its process. It protects this state, and by doing so guarantees that it is safe to interrupt Unison at any time: the system is therefore quite fault tolerant. More specifically, Unison guarantees the following:

- Each replica has either its original contents or its correct final contents (i.e., the values expected to be propagated from the other replica).
- The information stored in Unison's internal state is either unchanged or updated to reflect those parts of the replicas that have been successfully synchronised.

We discuss Unison's architecture, requirements and protocols in more detail in Chapter 9.

Some of the mechanisms needed to enforce these correctness conditions, particularly mutual exclusion, will now be discussed.

2.5 Low-level synchronisation

Many languages provide low-level synchronisation mechanisms for managing access to critical parts of a program. For example, consider a system that includes a database containing financial records: it may be desirable to provide synchronised read access to the database so that the data that are being read are guaranteed to be up to date when they are acquired. It will be essential to provide synchronised *write* access to the database, so that data cannot be lost in the writing process. These synchronisation mechanisms are needed to order, control and prevent conflicts in data access.

We discuss these problems in more detail in the following, under the broader category of *race conditions*. We also discuss some of the best-known and useful mechanisms for managing access, such as semaphores and mutexes.

2.5.1 Race conditions

Consider the example in Figure 2.1 where we have two programs A and B running at the same time. If both programs are working on the same x, what is the value of x when both programs have finished?

For now, we denote the starting value of x as x_0. If all three of A's instructions execute, then all three of B's, then the final value is x_0+3. However, there are 20 different interleavings of the instructions in A and B, and the possible

Program A	Program B
$a := x$	$b := x$
$a := a + 1$	$b := b + 2$
$x := a$	$x := b$

Figure 2.1 Two programs and a race condition

final values of x are $x_0 + 1$, $x_0 + 2$ and $x_0 + 3$. The final value clearly depends on the relative ordering of the instructions in the two programs.

This is an example of a *race condition*. We more formally define a race condition as *a critical dependency on the relative timing of events*.

Why are we concerned about race conditions? In this case, x is a single number and we have updated it as a single *atomic* (indivisible) operation. But what if x was a more structured type which requires many real CPU instructions to perform changes? In this case, we could end up with corrupted data. These types of errors can be hard to track down, as they manifest themselves as intermittent failures. Worse still, the error might only make itself apparent far away from the actual race condition.

To correct this, we require the three instructions in A to be treated as a single, indivisible instruction (and similarly for B).

2.5.2 Mutual exclusion

The classical answer to race conditions is the use of *critical sections* which are implemented by a *mutual exclusion (mutex)* mechanism. The idea of a mutex is to allow a process to *lock* a shared resource: this resource is exclusively used by the locking process until it *releases* its lock. This is typically implemented using a *semaphore*. Additionally, we require freedom from deadlock, that threads do not *starve* while waiting (i.e., they can eventually obtain a lock they require) and that there is no unnecessary delay.

⇝ *Deadlock* §2.5.3.3, p.21

```
initialise (s, r) is    s := r

signal(s) is            if    s > 0
                        then  s := s − 1
                        else  suspend

wait(s) is              s := s + 1
```

Figure 2.2 Pseudo-code implementation of a semaphore

2.5.3 Semaphores

Semaphores, attributed to Edsger Dijkstra [17], are a mechanism for enforcing mutual exclusion. They come in two variants:

binary semaphores, which protect a single resource; and

general semaphores, which protect a pool of identical resources (e.g., a collection of identical printers). General semaphores are sometimes known as *counting* or *integer* semaphores.

A binary semaphore is a special case of a general semaphore.

2.5.3.1 Implementation of semaphores. A semaphore uses a small non-negative integer as its state: we call this s. s is 0 when there are no resources left unlocked; s gives the number of free resources remaining.

There are three operations on semaphores: initialise, signal and wait. These are implemented in pseudo-code in Figure 2.2.

The first operation, initialise, sets the semaphore's 'value' to be the number of resources given (r): r is 1 for a binary semaphore.

The second operation, wait, is to be used immediately before a critical section. It has a conditional branch: if at least one resource is still available, then we decrement the count of free resources by 1 and continue. If there are no free resources, then the process that requested this resource is blocked: the underlying operating system suspends this caller.

Finally, wait is used when the critical section has been left: the resource that had been locked is now freed, so 1 is added to the count of resources.

So now our programs from Figure 2.1 can be written as in Figure 2.3, where y is a semaphore for the shared variable x.

2.5.3.2 Atomicity and semaphores. Look closely at the implementation of signal in Figure 2.2: what if *two* processes simultaneously try to call signal on the

Program A	Program B
wait(y)	wait(y)
$a := x$	$b := x$
$a := a + 1$	$b := b + 2$
$x := a$	$x := b$
signal(y)	signal(y)

Figure 2.3 Two programs using a semaphore

same semaphore? We will end up with exactly the problem we are trying to solve: both processes may end up proceeding into the critical section, even if only one was meant to.

We need to ensure that the signal and wait operations are processed atomically. For this, we need support from the underlying hardware and operating system. We have two common choices:

Test-and-set Some CPUs provide a single instruction that will both test the variable in a memory location and update it if appropriate. This is sufficient, since the CPU's instructions are atomic (at least, for our purposes).

Disable interrupts This prevents the operating system's scheduler timer interrupting in the middle of a critical part of the semaphore operations. Note that for most operating systems, disabling interrupts is a privileged operation, and we must remember to re-enable them afterwards.

Timer interrupts and multitasking §4.3.3, p.52

The two choices above will work on a uniprocessor system. However, multiprocessors systems have more problems:

- test-and-set must interact with the memory- and/or bus-controllers to ensure atomicity, otherwise other CPUs can interfere with the operation; and
- disabling interrupts on a single CPU makes no difference to the others, while disabling interrupts on all processors significantly affects performance.

A final option is the use of hardware-independent algorithms such as Peterson's algorithm, Dekker's algorithm or Lamport's Bakery algorithm (see exercise 2.5).

2.5.3.3 Deadlock. Suppose that a process P holds a lock giving it access to resource A. Process Q cannot use A until process P releases it. Suppose as well

that Q holds a lock giving it access to a different resource B. Thus P cannot lock B until Q releases it.

If P and Q each require both of A and B before they can make progress, then there is a risk of deadlock. If each holds one lock and neither is prepared to give up its lock, then they will never be able to proceed. This situation is called a *deadlock*. Deadlocks are unwanted in distributed programming since they prevent work from being done.

Deadlock in CSP §3.6, p.43

Ensuring that a distributed program does not have a deadlock is difficult. It is not possible, in general, to automatically detect deadlocks; the problem is undecidable. Designing programs to avoid deadlock is also challenging, and in general requires knowledge about the resource requirements for individual processes.

Deadlocks can arise in any program where the following conditions hold:

- The program includes resources that require exclusive access.
- Processes cannot be pre-empted (i.e., if a process holds a resource, it is the only process that can release that resource).
- Chains of waiting processes can arise, i.e., a process waiting for a resource held by a second process that is waiting for a resource held by a third process, etc.

Obviously, many useful programs satisfy these conditions. Avoiding deadlock requires careful use of synchronisation mechanisms, an awareness of how resources are being used, and analytic techniques to provide convincing evidence that deadlocks are, in most cases, avoided.

2.5.3.4 Semaphores, queues and priority inversion. An operating system might contain resources that many processes wish to access. If these resources are protected by a semaphore, then the operating system must maintain a queue of processes that are blocked on the semaphore.

In Section 2.5.3.3 several conditions were described as necessary for admitting deadlocks. One of these conditions was that processes could not be *pre-empted*, i.e., only the process holding a resource could give up its exclusive access to that resource. One of the approaches used to attempt to reduce the likelihood of deadlock is to use a *scheduler*. A scheduler is a program that allocates processes to resources, based on a *scheduling policy*. Two well-known and widely used policies are *first-come, first-served* and *round-robin*. In the former, processes are allocated to resources in the order in which their requests arrive to the scheduler. In the latter, each process is allocated a small (generally fixed) amount of time to access the resource, after which they release the resource. They may obtain the resource again, at a later time.

A problem with first-come first-served scheduling is that it can starve processes of resources: consider a process that holds on to a resource for a very long time; new processes that arrive and request the resource while the process is busy may not be able to access that resource.

Priority based schemes have been suggested as useful for helping avoid some of these problems. Each process (or task) is associated with a priority, and the processes are granted access to the resource in order of priority. This approach is appealing and not difficult to implement but also can lead to difficulties. A particular problem is that of *priority inversion*. This occurs when a low-priority process holds a resource required by a high-priority process. This may not cause problems with the correct operation of the program (i.e., the low-priority process eventually gives up the resource and the high-priority process can start its work). But complications may arise. The Mars Pathfinder is a particularly good example of the dangers that may develop due to priority inversion.[1]

There is no general solution to priority inversion problems, but typical solutions include disabling interrupts to protect critical sections, and making use of so-called priority ceilings, e.g., assigning a very high priority to the operating system, which is responsible for locking and unlocking mutexes.

2.5.4 Monitors

A *monitor* is another commonly used synchronisation mechanism in computer software. Like semaphores and mutexes, a monitor provides synchronised access to shared resources, and can be implemented either in hardware or using variables. The main difference between a monitor and the synchronisation mechanisms we have seen so far is the level of abstraction: monitors effectively encapsulate the low-level synchronisation details that must be considered when using semaphores or mutexes.

↝ *Ada protected object* p.71

A monitor, associated with a shared resource, consists of four parts:

- procedures that support the interactions with the shared resource;
- a lock, in order to provide mutually exclusive access to the resource;
- variables for the resource;
- *conditions*, capturing what is needed to avoid race conditions.

Here is a small example of a monitor, written in Eiffel-like syntax. See [19]

↝ *Eiffel* §8.6, p.149

[1] The Pathfinder experienced total system resets, as a result of priority inversions involving an interrupt, a communications task and a low-priority meteorological thread. A watchdog timer would go off, generating the system reset.

for more details on Eiffel. The monitor construct encapsulates a resource — in this case, a shared bank account— and provides exclusive access, via its operations, to this resource. Mutually exclusive access is needed so that all operations on the bank account obtain the most up-to-date data stored in the account.

```
monitor BANK_ACCOUNT
 creation make
feature
  balance : INTEGER

  make is do balance := 0 end

  withdraw(amount:INTEGER) is
  do
    if amount < 0 then
      io.putstring("Amount cannot be negative.")
    elsif amount > balance then
      io.putstring(" Insufficient funds.")
    else
      balance := balance − amount
    end
  end

  deposit(amount: INTEGER) is
  do
    if amount <0 then
      io.putstring("Amount cannot be negative.")
    else
      balance := balance + amount
    end
  end
end
```

The bank account initialises its balance to zero. Clients of the bank account can then either withdraw funds (via operation withdraw) or deposit funds (via operation deposit). Implicit in this monitor is the lock, which can be held by one operation at a time. An operation executes only when it holds the lock; otherwise it is waiting.

The above monitor must also describe validity properties; in this particular case, a property must be expressed that says that the account balance is up-to-date, and expresses the results of all previously executed operations.

⇝ *Condition variables* §5.2.3, p.68

Monitors typically come with *condition variables* that can be used to signal tasks or processes about interesting events. These events can thereafter be used to more precisely constrain synchronisation. If an operation in a monitor needs a condition to be true before proceeding (e.g., that amount>0) then it *waits* on a particular condition variable. By waiting, the operation gives up the lock — it thereafter cannot be executed. Should another operation

subsequently cause the condition to be true, then it can signal this by *notifying* on the same condition variable.

```
withdraw(amount:INTEGER) is
do
    if amount < 0 then
      io.putstring("Amount cannot be negative.")
    elsif amount > balance then
      wait(nonzero_balance)
    else
      balance := balance − amount
    end
end

deposit(amount:INTEGER) is
do
    if amount <0 then
      io.putstring("Amount cannot be negative.")
    else
      balance := balance + amount
      notify(nonzero_balance)
    end
end
```

2.5.5 Rendezvous

So far, we have looked at race conditions and prevention using mutual exclusion mechanisms such as semaphores and monitors. Now we arrange for two concurrent processes to interact with each other in a controlled way. Sometimes, we want to arrange that two active processes synchronise, perhaps as part of a commitment protocol, or to pass data from a producer to a consumer. This type of synchronisation is called a *rendezvous*.

Unless all parties to the rendezvous are ready at exactly the right time, at least some will have to wait. So if A and B are to synchronise at a particular point in their processing, one or the other will be ready first, and will wait until the second is ready.

We will see examples of rendezvous in the CSP process algebra later where we illustrate a simple producer-consumer example. Then we return to this producer-consumer example by implementing it in Ada.

⇝ *CSP* §3.4.1, p.37

⇝ *Ada tasks* §4.5, p.58

2.6 Timing and real-time systems

A real-time system is one whose correctness depends not only on the correctness of a result, but also on the time when the result is produced. Sometimes such systems are classified as 'hard' real-time or 'soft' real-time, where hard real-time systems are incorrect (and the result has no value) if a deadline is missed, whereas the result still has a value, albeit decreasing, after the deadline in soft real-time systems.

Real-time systems are not necessarily fast systems: they may take a long time to perform a task and produce a useful result. This is still correct behaviour if the required deadline is met.

For a fuller treatment of real-time systems (particularly applied to the Ada language) see the classic text [8] by Burns and Wellings.

2.7 Dependability

A distributed system, like any other computer system, has a specification that it is expected to satisfy. When we attempt to satisfy this specification, we will have a number of *quality attributes* that we will have to trade off in the course of constructing a design and implementation. Of paramount importance in a distributed system is the idea of *dependability*. Dependability is a complex notion but for distributed systems it primarily focuses on a requirement for the system to be tolerant to *faults*. A fault causes some kind of *error* which prevents the system from meeting its specification. A system that does not meet its specification is said to *fail*.

Dependability, more precisely, can be broken down into several additional attributes like the following:

- *Availability:* an available system is one that is always ready to be used and to deliver services. A highly available system is one that will be ready to deliver service at almost any instant.
- *Reliability:* a reliable system is one that can run continuously, for some period of time, without error. While this sounds similar to availability, it is defined in terms of errors within a time period. Thus, a highly reliable system might meet its specification continuously for a long period of time, e.g., months or years.
- *Safety:* a safe system is one which does nothing catastrophic even in the presence of temporary faults.

– *Maintainability:* a maintainable system is one which is easy to repair in the presence of faults. There are tradeoffs between maintainability and, for example, availability: if a system is easy to maintain it may be more difficult to keep it always available.

Building a dependable distributed system generally means making it *fault tolerant*, i.e., ensuring that the system provides services even in the presence of faults. Faults are thus anticipated and mitigation is built in to a fault-tolerant system. This does not necessarily mean that if a fault arises, the same level of service can be provided as when there are no faults; degraded levels of service may also be provided.

2.7.1 Types of faults and failures

Faults are generally classified into three types. *Transient* faults occur once when an operation is carried out; when the operation is repeated the fault does not reoccur. *Permanent* faults occur and remain until the source of the fault is repaired or replaced (e.g., a burnt-out component). *Intermittent* faults are neither transient nor permanent; they occur, disappear, and then return of their own accord. These are the most difficult faults to deal with because of their inherent unpredictability.

Similarly, failures are classified into a number of types including

– *Byzantine:* the most serious type of failure, where a system may produce arbitrary results at arbitrary times.

– *Crash:* a failure where a system stops operating but had been operating according to its specification before stopping. Once a crash has occurred, the system does not return to operation. An operating system crash, from whose recovery requires rebooting the machine, is a good example of this type of failure.

– *Omission:* such a failure occurs when a system fails to respond to a request. For example, a connection between peers may have disappeared, removing the means for one peer to respond to the requests of another. Another good example arises when a system enters a 'busy loop' in which requests are not processed: a response will never be received in this case.

– *Response:* a response failure (or *commission failure*) is related to omission failures; a response is received to a request, but it is incorrect. For example, a request to add a book to a shopping cart in an online store, which results in the wrong book being added, would be a response failure.

2.7.2 Responding to failure

Ideally, we would like to build a system that is reliable (and satisfies its specification) from the start. Using good software engineering practices, such as focusing on simple designs, eliminating redundant code, using modular designs, and frequent testing, will go a long way towards this. However, distributed systems do not operate in a vacuum: they interact with users, other systems and the greater outside world (e.g., a network, or the Internet). Things outside of the system are clearly out of our control and failures that originate outside of the system may still need to be managed from within the system. This requires the system to be fault tolerant.

The key to making a system fault tolerant is to mask failures, to prevent them propagating through the system and to catch them as early as possible. The most well-known and practised technique for fault tolerance is to use *redundancy*. Redundancy may involve adding extra equipment, components or software to guarantee that a failure does not propagate and a specification is satisfied. Redundancy of this nature is wide-spread in avionics systems (e.g., replicated sensors on aircraft, and/or replicated processes for analysing sensor data) and in electronics systems. Redundancy can also arise by running processes multiple times to obtain a result: this is particularly useful for dealing with intermittent faults: if a fault arises as a result of running a process, we might try to run the process again, perhaps thinking that the environment has changed and the fault will not reoccur.

2.8 Server types

This chapter has examined low-level mechanisms for synchronisation and issues regarding dependability. When producing a system that serves clients, a *server*, there are two main types, *iterative* and *concurrent*. In each case, we assume that the requests from clients can be represented as a single first-in, first-out (FIFO) queue.

TCP server §5.5.1, p.75

Iterative or sequential servers are the simplest case. They are easy to implement: a simple loop takes the first job from the queue (or waits until a job is available if the queue is empty) and processes it. A side-effect is that the need for mutual exclusion does not arise as there is no concurrency. However, if the processing is time-consuming, then clients may have to wait for a long time: consider examples such as FTP servers or HTTP servers that may deliver large files to users. Iterative servers are best suited to simple tasks that are guaranteed to be completed quickly.

Concurrent servers operate by creating a child server to handle each request. These can be very simple provided that the operating system has appropriate system calls. In Chapter 4 we will see a number of primitives (basic commands or functions) for creating processes and threads, and communications between them. Mutual exclusion and all the other issues associated with concurrency must be considered for such servers.

Forking server §5.8, p.91

There are variants in between: we may impose a maximum on the number of concurrent children running at any one time. A common pattern seen in web servers (e.g., Apache) is to create a fixed number of children when the main server starts. These children then process a number of jobs before terminating and being replaced by new children. This is known as *pre-forking*. (The termination and replacement of child processes is to cope with programming errors that might result in memory leaks, for example.) The type of concurrency may be varied or even combined: a process may fork further processes and each child process might itself create threads.

Threads §4.3.4, p.54

We can consider further factors of servers: the jobs may be connection-oriented or connectionless. The former involves multiple messages between the client and server over a reliable transmission link provided by the operating system (e.g., TCP). The latter has lower overhead, but the application (both client and server) must cope with potential loss of messages (e.g., UDP).

Finally, a server may be stateful or stateless: a stateful protocol involves a conversation between the client and server where earlier messages constrain or allow future messages. A stateless protocol is usually simpler. We can add variations to these servers, such as caching credentials or recently computed results.

2.9 Clusters, load-balancing and Grids

A computer *cluster* is a group of computing devices that are loosely coupled together to provide robust, reliable functionality typically at greater speeds than is possible with single devices. It is the job of the operating system (see Chapter 4) to hide from the applications the fact that they are executing on a collection of computers rather than a single computer.

Generally, clusters come in several varieties, including:

– *Load-balancing* clusters, which aim to distribute workload (e.g., jobs) relatively evenly across devices in the clusters. This is usually accomplished by having one or more front-end servers that are load-aware, and that are responsible for distributing workload across servers on the back-end. Such clusters are sometimes called *server farms*, a model also used by

Google for its web search facilities.

- *High-performance* clusters, which split jobs up across a number of computing devices. These clusters are aimed at increasing performance (e.g., for large numerical computations) and are often used for scientific computations. A popular example of a high-performance cluster is a *Beowulf* cluster [72], where individual nodes in the cluster are running Linux as well as software to support parallelism, such as PVM.
- *Grid* computing is a variant of cluster computing where the individual devices in the cluster may be owned and operated by different individuals and organisations, which may not totally trust each other. Additionally, Grids tend to be more heterogeneous than other forms of clusters. Grids also aim to provide high performance and high availability, like other types of clusters, but are ideally suited to jobs which can be split up into many independent parts that do not need to share data.

A variety of software exists to help construct clusters, ranging from commercial products to open-source technology such as Sun's GridEngine.

2.10 Summary

The exploration of the fundamental concepts of concurrency in this chapter serves as an essential prelude to understanding the fundamentals of distributed programming. We saw some of the most important architectures for concurrent systems, which are widely used in a variety of applications, including computer games, version control systems, chat systems and file sharing systems. At the heart of these architectures is the notion of name and naming, which we discussed in some detail, exploring the different variants of names that are pertinent in building concurrent and distributed systems. We then considered some of the key problems in concurrency, such as resource sharing, synchronisation and race conditions, as well as some of the principles and mechanisms that can be used to manage these problems: mutual exclusion, monitors and semaphores. We then discussed issues related to these key concurrency problems, particularly the issue of timing, and when systems need to meet deadlines and achieve goals under temporal constraints, and the more general issue of dependability of concurrent systems. This was concluded with a discussion of some key concepts used in building large-scale concurrent systems, particularly servers, clusters and Grids.

Further reading on the development of clusters using open-source software can be found in [37]. The history of the development of clusters is well

presented in [52].

In the next chapter, we will refine our understanding of the basic concepts of concurrency by considering several *models* of concurrency, which will allow us to describe abstract concurrent systems, reason about them and experiment with different mechanisms for solving the fundamental problems of building concurrent systems.

EXERCISES

2.1. What are the key differences between a client-server and a peer-to-peer architecture? Can you think of situations in which one might be preferred over the other?

2.2. Can a client-server architecture be used to support or implement a peer-to-peer architecture? Explain why this is or is not possible.

2.3. What do you think is meant by the phrase *busy waiting*? What might constitute non-busy waiting?

2.4. A multi-semaphore allows the two primitives wait and signal to operate on several semaphores simultaneously. This allows concurrent systems to acquire and release several resources at once. The *wait* primitive for two multi-semaphores S and R can be described using the following pseudocode:

```
from
   until  (S<=0 or R<=0)
loop ;  end;
S := S−1;
R := R−1
```

Describe how a multi-semaphore can be implemented using (more than one) regular semaphores.

2.5. Here is a pseudo-C implementation of the so-called *Bakery* algorithm. Does it solve the critical region problem, i.e., does it allow a single process at a time access to the critical region? Explain your answer.

```
1  /* Shared data */
   int number[n];  /* All  initially  0 */

   /* Each process Pi (i=0..n−1) looks as follows */
5  number[i] = max(number[0],number[1],...,number[n−1])+1;
   for(j=0;  j<n; j++){
      while((number[j] != 0)  && number[j]<number[i] && j<i) ;
```

```
    }
10  /* Critical region */
    number[i]=0;
```

2.6. What are the necessary conditions for unbounded priority inversion to occur in a priority-based scheduling system? Give an example of priority inversion with three tasks with three different priorities.

2.7. Describe the characteristics and the behaviour of a *monitor*, including discussion on the applicability of condition variables.

2.8. Extend the monitor construct to allow nested calls. In other words, a method executing within a monitor can make a call to a method in a different monitor. One issue to consider is what happens to mutual exclusion locks. For example, if a method in monitor A makes a nested call to a method in monitor B, should it lose the lock on A?

2.9. Choose a system with dependability requirements, like an airplane engine controller, software for controlling a medical device (e.g., a pacemaker), or a point-of-sale system. What are the important dependability requirements for the software you have chosen? How would you argue that any implementation of this system is adequately dependable?

2.10. What are some of the different kinds of faults that can manifest themselves in systems?

http://www.springer.com/978-1-84628-840-1

Practical Distributed Processing

Brooke, P.J.; Paige, R.F.

2008, XIV, 262 p. 24 illus., Softcover
ISBN: 978-1-84628-840-1