A
Exercises: Hints and Comments

Chapter 1: What is Distributed Processing?

1.1 Suggest several kinds of resources that might be shared in a distributed system. For each resource, describe one challenge that may be encountered in sharing the resource.

Ans. A database, a mail server, a password server. Some examples of challenges: security for the password server, scalability for the mail server, fail-safe for the database.

1.2 Suppose that you have been given a server (e.g., a web server) and wish to write a client for it. Describe several ways in which the server may fail. Briefly explain how you might mitigate these failures in your client (if, indeed, it is possible to do so in the client).

Ans. The server may contain a fault and as a result it may crash or raise an exception. Another failure might be that it delays sending critical messages to a client, or sends a repeated message where one was not expected. Clients should probably filter any exception information that isn't relevant to them, i.e., exceptions that they cannot handle should be ignored. Clients may ignore repeated messages that are irrelevant. Message delays are more challenging, particularly in settings where a real-time response is required. Some clients attempt to predict what the message might be, and a priority-based approach may be used to indicate that a client is being 'starved' of important messages.

1.3 Give an example of a client-server application that you are familiar with. Is there any advantage to making it a peer-to-peer application? If so, what would the nodes in the application be?

Ans. A multiplayer game is often a client-server application. Many other examples exist. Making a client-server application a P2P application depends on a number of constraints; a multiplayer game *could* be made P2P but it may be difficult to achieve usable response times and frame rates. It is questionable whether it is really appropriate to use P2P architectures for games, as generally there are elements of trust to consider: do we trust all players (and hence, all clients)?

1.4 Consider a multiplayer game, supporting many thousands of players. The game provides a number of servers that are connected somehow in a network architecture. Suggest an architecture for these servers, and explain the benefits and disadvantages of your architecture.

Ans. This is an open question with many answers. A key issue to discuss is tradeoffs. Another key issue is what kind of game is being supported: one has different requirements for a real-time strategy game, versus a first-person shooter, versus online gambling where real money is tangled up in gameplay. Most obvious architectures have been tried. Most often clusters of servers (or server farms) are used. These are typically connected in a client-server architecture. There are attempts at more distributed architectures (sometimes using distributed hash tables and clever algorithms for load balancing and data consistency), including peer-to-peer. It is not yet clear whether these architectures will provide the needed levels of performance for commercial use.

1.5 Consider a Java object that provides a method called foo(). Suppose that you want to make this object available over the network, on a remote computer. Discuss the difficulties that arise with allowing clients, on a separate computer, to call the foo() method. In other words, what might you need to do to allow such *remote* calls to foo()?

Ans. Provide the means to serialise arguments and distribute them to the remote object; type checking of serialised arguments; security mechanisms to ensure that the remote object allows the client to access foo(); exception handling mechanisms, in case the remote object fails during execution — this last point is particularly difficult!

1.6 When you move around the country, your mobile phone calls are handled by different processing units, depending on where you are. Is this how your calls would be handled if you travelled to a different country? Are there additional issues to deal with in this situation?

Ans. In principle, it is the same, but additional details concerned with transferring to a different network must also be handled. This may involve negotiation of different protocols or authentication schemes and will certainly involve connections to different billing schemes.

1.7 What are the differences between security and dependability?

Ans. Security is specifically concerned with managing risks and mitigating threats associated with using a distributed system; dependability is more general, and is also concerned with issues such as availability, reliability, robustness and survivability.

1.8 Contrast a distributed system with a concurrent system. What are the main similarities and differences?

Ans. A distributed system runs across a network on multiple units of hardware. A concurrent system runs multiple computations at the same time; these may or may not be distributed across a network.

1.9 Consider, again, a mobile phone network, and suppose that when you are attempting to make a call, the network node that is handling your call fails. Suggest some strategies for dealing with this failure, that will (ideally) allow you to make your call.

Ans. An obvious strategy is to attempt to pass handling for the call to a nearest neighbour. The neighbour needs to be found by a sensible algorithm that takes into account the range of a mobile. Sometimes a new handler cannot be found; in which case, it would be sensible to queue/buffer the call details, e.g., on the phone itself, and to indicate to the caller that a connection cannot be made at the moment and that they should retry at a suitable time in the future.

1.10 Suppose that a failure occurs in a distributed system, and an *exception* is raised in component C. The exception handler for C is located in component E, but suppose also that E failed and crashed fatally 20 seconds prior to C's exception. What can be done to process the exception in C?

Ans. This is a hard problem, that of distributed or asynchronous exception handling. There is no good general answer to this, beyond either (a) allowing the main thread of control to process the exception; (b) attempting to restart C; or (c) trying to find a proxy to process the exception. All of these have their own disadvantages.

Chapter 2: Concepts of Concurrency

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?

Ans. There are several key differences: the client-server architecture centralises much of the processing in one (or more) servers, whereas this processing is distributed in the P2P model. P2P probably allows easier addition of new nodes, whereas client-server can be hard to use to add new servers (though this depends on the system). P2P introduces some challenges with data consistency. Other attributes can and should be discussed. Management and control can be substantially more complicated with a P2P architecture versus a client-server architecture.

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.

Ans. This question can be interpreted in many ways. One reasonable interpretation is to ask whether the programming and design idioms for client-server can be used to build a P2P architecture. The answer in this case is yes; one can use client-server primitives to enable P2P communication. Effectively, when two peers exchange data, this can be viewed as two client-server communications.

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

Ans. Busy waiting involves a process repeatedly checking whether a condition holds (e.g., that a boolean variable is true). If the condition is true then the process moves on to do new work, but if the condition is false the process continues to check. Thus, busy waiting delays execution for some time. This is sometimes called spinning, because clock cycles are generally wasted. Non-busy waiting would involve temporarily pausing a process while it waits for a condition to hold, and making use of the CPU cycles for other tasks. Busy waiting can almost always be avoided, e.g., by putting threads to sleep so that they consume no CPU cycles, using signals. But busy waiting is sometimes really needed in hardware driver programming, in particular because implementing lots of interrupts is expensive and impractical.

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.

Ans. The basic way to do this is to have a sequence of regular semaphores that are acquired and released in order. Thus, when you want to acquire a multi-semaphore you order the semaphores (this will be implementation dependent) and try to lock each in order. If any semaphore can't be acquired (e.g., due to timeout or interrupt) then all must be released. On release, each semaphore in the sequence is unlocked/released in order. Generally, multi-semaphores define a comparison operator on semaphores, e.g., based on some kind of semaphore ID.

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;
```

Ans. It is instructive to compare this with Lamport's implementation. Lamport includes an additional variable called choosing, which indicates which process is trying to enter the critical region. This variable is omitted in the above code. As it turns out, the above example violates mutual exclusion. Two processes P and Q reach line 7 at the same time. Assume that both read number[0] and number[1] before the addition (+1) takes place. Then assume that Q finishes in line 7 (and assigns 1 to number[1]), and P blocks before it assigns to variable number. So then Q gets in to the while-loop and enters the critical region. While there, it blocks; then P unblocks and assigns 1 to number[0] in line 7. It then enters the while-loop. Consider the iteration when j=1; P can now enter the critical section, thus violating mutual exclusion.

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.

Ans. The second part is more interesting. Consider three tasks H, L, M with priorities high, low, and medium respectively. Tasks L and H share a resource. Task L takes the resource, and shortly afterwards H becomes ready to run, but of course it must wait until L is finished. Before L finishes, M also becomes ready to run, and it therefore pre-empts L. While M runs, task H, the highest priority task in the system, has to wait.

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

Ans. The basic behaviour of a monitor is straightforward. It takes a lock on a resource and holds it until a condition is satisfied. A monitor may also have an invariant which describes assumptions needed to avoid race conditions. With respect to condition variables, these are used to allow processes to signal each other about interesting events. So when a function in a monitor needs a condition to be true before proceeding, it waits on the condition variable associated with this condition. In doing so, it surrenders the lock on the shared resource. If another process later causes the condition to be true, then this may notify, using the condition variable, any process waiting for the condition. A notified process gains the lock and then proceeds.

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?

Ans. There is really no right answer to this one. There are probably a couple of sensible approaches. (i) Keep the lock, and thus block waiting processes potentially for a long time. This could lead to deadlock. In particular, if we are implementing monitors using signals, we should consider the semantics of the signal operation; it is probably best to prioritise waiting nested calls over non-nested calls, and a waiting process is resumed in preference to a signalling process. (ii) Release the lock; how do we then leave the monitor in a consistent state when the nested call is made? We must also consider how the process regains the lock of A when the nested call is completed. The current approach potentially unblocks monitors more quickly.

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?

Ans. This is a general question that asks the reader to think about arguing dependability and justifying it. What we might look for in a good answer is a discussion on how one presents and documents an argument. Of particular importance here is traceability. A good argument will draw on the evidence needed to justify that a dependability requirement is satisfied, and will clearly show how this evidence traces back to individual (or collective) requirements.

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

Ans. Omission faults, i.e., a component is not performing an interaction it was specified to perform (e.g., a crash, a periodic omission of a specified interaction, a timing fault), an assertive fault, i.e., where interactions were not performed up to specifications (e.g., sending a float instead of a short, sending a bad value), and arbitrary faults (e.g., improbable sequences of events, deliberate actions by intruders, byzantine faults).

Chapter 3: Models of Concurrency

3.1 Explain why concurrency models (like state machines or process algebras) are helpful in designing concurrent systems. When might a concurrency model prove awkward or difficult to use in designing systems?

Ans. Concurrency models help us to precisely (and in many cases, mathematically) describe concurrent systems from an abstract perspective, omitting details that may hinder our understanding of the concurrency aspects of a system. In many cases, we can reason about our concurrency models and prove properties about them (e.g., deadlock freedom). Concurrency models can help us detect defects, omissions, flaws and errors prior to implementation. A concurrency model may be difficult to use if the system of interest also involves substantial non-concurrent

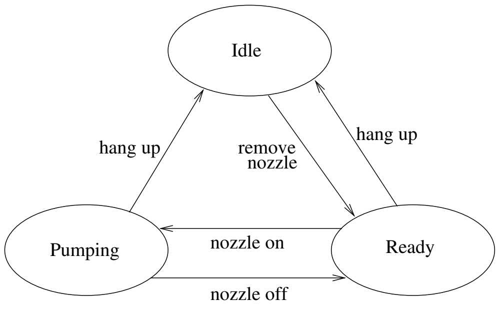

Figure A.1 Petrol pump state diagram

elements that are tightly coupled with concurrent aspects, thus complicating the concurrency model. Moreover, if many different concurrency models are needed, these may need to be merged and integrated (and checked for consistency) to be useful.

3.2 Draw a state machine for a simple petrol pump. A pump is either idle, ready, or pumping petrol. Pumping commences when the handle on the nozzle of the pump is squeezed, and stops when the handle is released. When the nozzle is removed from the pump itself, it is ready to be used. When the nozzle is hung up on the pump, the pump is considered to be idle.

Ans. There are several ways to solve this problem, but Figure A.1 illustrates one (state machine) solution. An interesting extension is to add handling of payment details.

3.3 In the CSP example on page 44, how can the process Q be modified so that deadlock does not arise?

Ans. Modify the order of locks: if both P and Q always lock A *then* B, this example is safe from deadlock. You might want to use FDR2 or CSPsim to demonstrate this.

3.4 The critical section problem was discussed in Chapter 2. In this problem, two or more processes must mutually exclusively enter a critical region to do work. The following pseudocode is proposed to solve the critical section problem.

```
var
  integer turn := 1;
  boolean flag1 := false;
  boolean flag2 := false;

process P1
```

```
begin
  while true do {
    flag1 := true;
    turn := 2;
    while flag2 and (turn = 2) do skip;
    (* Critical Section for process P1 *)
    flag1 := false;
  } end;
end;

process P2;
  (* similar to P1 but setting flag2,
     setting turn to 1 and checking flag1 in while loop *)
```

Write a CSP program for this algorithm. How might you actually demonstrate that the CSP program guarantees mutual exclusion?

Ans. The difficult part here is encoding the pseudocode correctly into the model checker's language. Once that is done, the model checker can solve the problem for you.

3.5 Discuss what a concurrency model allows developers to accomplish. Explain what a concurrency model does *not* allow developers to do.

Ans. The first part is discussed in the answer to the previous question: analysis, focusing on a specific set of system attributes, etc. The second part is more interesting. A concurrency model does not always provide an implementation, nor does it always help in understanding the relationship between concurrency aspects of a system and non-concurrent aspects. Moreover, some concurrency models are non-trivial to transform to executable code.

3.6 Briefly explain the key differences between state machines and process algebras for modelling concurrent systems. Can you think of a situation where you might prefer to use state machines instead of process algebras?

Ans. At a superficial level, we might say that the former is graphical while the latter is textual. They each have different semantic models. State machines are generally executable (and simulators exist for many dialects). Process algebras can be simulated, typically by generating a state machine representation. Process algebras generally support a notion of refinement and/or bisimulation; refinement for state machines can also be defined. In general, process algebras and state machines are very similar from the perspective of their general capabilities.

3.7 Consider the example Promela program in Section 3.3 (page 35), which splits messages between two output streams. Write a Promela program to merge the two streams into one. Can you guarantee that the order of messages after merging is the same as prior to splitting?

Ans. This is quite straightforward. A sensible solution would be:

```
proctype merge()
{
   short msgs;
   do
   ::  if
       :: large?msgs
       :: small?msgs
       fi ;
       in!msgs
   do
}
```

i.e., we take any input off any channel and output it thereafter. Clearly this does not guarantee that ordering is preserved, as the procedure can nondeterministically select from either large or small channels.

3.8 Consider the previous question; how can you modify your Promela program to ensure that ordering is preserved, i.e., that messages, when merged, are kept in the same order as they were before splitting?

Ans. This is a little tricky. One approach is to force an order on the output process (i.e., when data are split) and to use the same order on input. For example, we might add a counter to each message, so that when it is output on *msgs* its order number is also included. Then, in *merge*, we could add a counter, and each time we read data from the large and small channels, we must read the message that has an order number corresponding to that of its counter. This requires synchronisation between where the counters start in both merge and split .

3.9 The JavaSpaces example in Section 3.5.1 (page 42) did not update the counter indicating how many times a tuple entry has been read. Write a JavaSpaces class that provides this functionality on take.

Ans. This is relatively straightforward; the basic idea is to again construct an empty template and apply space.take, apply the increase method of class Message, and then write the resulting entry back to the space. Note that you must use *take* instead of *read* since you need to increase the counter outside of the space. [23] has some additional information on this.

3.10 Using any language you like, write a simple program or specification with two functions/routines, t1 and t2, such that if these functions are called by two different threads, they may generate a deadlock. Explain in a couple of sentences how the deadlock could be avoided for your program.

Ans. The point of this question is to make concrete what a deadlock really is. Here is a little program in Java (but of course other languages could be used):

```java
class DeadlockExample {
  Object o1 = new Object();
  Object o2 = new Object();

  public void t1(){
    synchronized(o1) {
      synchronized(o2) {}
    }
  }

  public void t2(){
    synchronized(o2) {
      synchronized(o1) {}
    }
  }
}
```

To help avoid the deadlock, you need to make sure that you never hold a lock on both o1 and o2 at the same time. If both locks are needed, then you must make sure they are always obtained in a consistent order.

Chapter 4: Concurrency in Operating Systems

4.1 Summarise the objectives of an operating system.

Ans. One objective is improving convenience of use, i.e., providing an easier interface for users. Another is for managing resources. A third is to make it easier and more reliable to maintain systems and software.

4.2 Research the structure and components of the Windows XP operating system. Determine the important components and how they connect. Draw a UML diagram of the basic structure of Windows XP.

Ans. It is useful to start this by looking at Microsoft's own XP Technical Overview. Some of the basic components include a hardware abstraction layer, kernel, executive, virtual memory manager, I/O manager and security subsystem.

4.3 Run the example fork code on page 55. Why does PPID for the child eventually become 1?

Ans. The parent process ends before its child, but every process must have a parent. In this case, the init process (with PID 1) 'adopts' the orphaned child process.

4.4 A process is in its critical region, managed by a mutex. The process itself generates a fatal error which causes it to be killed. How could this affect other processes? Suggest how the operating system might mitigate this problem.

Ans. A key problem here is whether the lock on the critical region is ever released; if it isn't, then other processes will starve. The OS could help to deal with this by monitoring signals that cause processes to die unexpectedly; this could then trigger an urgent interrupt of the lock. In real-time programming, this is sometimes called a *duel*. Another problem is whether the process in the critical region left the system state inconsistent: it is not simply a matter of providing an exception handler, because there is nothing really available to handle that exception. The OS may take responsibility in running a generic exception handler to deal with this kind of situation.

4.5 In most dialects of Unix, processes are given priorities, and these priorities are reordered from time to time. Research how dynamic re-allocation of priorities works, and explain any benefits or difficulties with this approach.

Ans. Research question. A good place to start reading is the O'Reilly book *Understanding the Linux Kernel* by Bovet and Cesati.

4.6 Consider the following C fragment (**WARNING:** do **not** execute this program on a shared machine for which you do not have responsibility!).

```
while(1) fork ();
```

Describe the dangers associated with this program, and propose a means to mitigate this danger.

Ans. This is, of course, a fork bomb. It is a denial-of-service attack. The attack relies on there being a limit to the number of processes that can run simultaneously. It attempts to saturate the available space in the process list held by the OS. It will slow down the system. The only solution to a fork bomb is to destroy all its instances, e.g., *kill* or rebooting. Prevention is by limiting the number of processes that a single user can own. Unix systems typically provide a limit, controlled by ulimit. Limiting the number of processes that a *process* can create won't prevent a fork bomb. Why?

4.7 Here is a simple C program that makes use of fork().

```
main(){
  int i=0;
  int childpid;

  printf("Parent PID is %d\n", getpid());
  while(i<3){
    childpid=fork();
    if(childpid!=0)
      printf("%d: childpid: %d\n",i,childpid);
    i++;
  }
}
```

What output might be generated from an execution of this program? In particular, discuss why, when this program is run on a Linux machine, the command line prompt might appear before the output from the printf statements.

Ans. The prompt can be displayed as soon as the above root process has finished; it need not wait until the children have finished. Thus, output may not be exactly as you expect.

4.8 Write a Pthreads program showing *interference* between two threads sharing a variable.

Ans. We can implement such a program by using some of the basic Pthreads functions we used in this chapter. Here is an example.

```
#include <stdio.h>
#include <pthread.h>

/* Shared variable */
int count;

/* The routine that tampers with the
   shared variable. */

void *tamper() {
  int i;
  for(i=0; i<5000; i++)
    count ++;
}

main(){
  pthread_t thr;
  do {
    count = 0;
    pthread_create(&thr,NULL,tamper,NULL);
    tamper();
    pthread_join(thr,NULL);
  } while (count==10000);
  printf("%d\n", count);
}
```

The thread thr starts executing the function tamper(). At the same time, the main thread executes tamper(), and when we execute pthread_join(), the loop has executed twice.

4.9 Write a Pthreads program that takes a number n as input, and creates n threads, each of which prints out a message and its own thread ID. Demonstrate thread interleaving by making the main thread sleep for a couple of seconds for every few threads it creates.

Ans. This is straightforward and involves use of pthread_create(). It is interesting to experiment with different sleep times, different periods between sleeping, and different values of n. It is also useful to look at gcov() to evaluate the amount of time spent on thread creation.

4.10 What does deadlock mean in terms of a set of two or more Ada tasks? Consider the following program definition of three Ada tasks.

```
task author is
  entry writer;
  entry reader;
end author;

task printer is
  entry typesetter;
  entry binder;
end printer;

task artist is
  entry inker;
  entry colourist;
end artist;
```

author invokes only printer.typesetter and printer.binder. printer invokes only author.writer, artist.inker and artist.colourist. Assume that artist invokes only author.reader. Can these tasks deadlock? Explain your answer.

Ans. A general answer to what constitutes deadlock in Ada tasks is that there is a cycle of two or more tasks, where each task simultaneously reaches the point of attempting to rendezvous with the next one in the cycle. To answer the second part, we can draw a rendezvous graph. There is a directed cycle in the above example, suggesting that deadlock is possible. In particular, author could be waiting for a rendezvous with printer.typesetter, while printer is waiting for a rendezvous with author.writer. Obviously this is a deadlock.

Chapter 5: Interprocess Communication

5.1 Can you prove that philo.adb in Section 5.3 (page 71) deadlocks or never deadlocks?

Ans. philo.adb will deadlock. Suppose that each philosopher picks up the left fork. Then none can ever pick up the right fork — deadlock. There are several solutions: one is to require that the forks are picked up in a strict order. In our example, we would require them to pick up fork A before fork B, and so on.

5.2 Section 5.4 introduces a number of C system calls and functions, such as socket(), recv() and inet_pton(). Locate the Unix manual pages for these functions in (manual) sections 2 and 3.

Ans. This is a simple application of man.

5.3 What possible drawbacks are there to the server like the one in Section 5.8 (page 91)? In what ways could a malicious user abuse the system?

Ans. A malicious user could harm the system by resource exhaustion. By starting many inbound connections, the use of fork() will create as many processes as are permitted by the operating system.

5.4 The example server in Section 5.8 (page 91) does not use fork() safely. What else should it do?

Ans. The man page for fork() tells us that -1 is returned if the child process cannot be created.

5.5 In Section 5.9.1 (page 95), multiple messages from the server to the client are sometimes concatenated. Why is this?

Ans. Our examples have removed the newlines. So all that is received is a simple stream of characters: there is no other structure imposed. Modifying the server to send a newline after each string might be appropriate in some cases (see skt4−server2.c).

5.6 Modify skt4−client.c and skt4−server.c to use select () to check that they can send their messages. What happens if skt4−server.c tries to send back a large number of anagrams at once?

$\boxed{\mathcal{EG}}$ skt4-server3.c

$\boxed{\mathcal{EG}}$ skt4-client3.c

Ans. The problem here is that server may not be able to send while intermediate buffers are full. This depends on the socket in the server: if it is blocking, then it will simply wait until it can send the message. If it is non-blocking, then send will eventually fail. The examples skt4−server3.c and skt4−client3.c illustrate this.

5.7 Recall your answer to Exercise 4.8, where two threads were interfering with each other. Modify your answer so that threads cooperate in changing the variable, i.e., use Pthreads' mutual exclusion mechanisms.

Ans. This is straightforward. We introduce a Pthreads mutex and within the function tamper() we acquire the lock prior to updating count, after which we unlock the mutex.

5.8 Write a Pthreads program as follows. It accepts two kinds of command line parameters: a single number, which indicates the program should run with exactly two threads; and a pair of numbers, e.g., Program 5 2. The first number is an argument, the second the number of threads. The first number is the largest number tested for primality by the program. The program tests all numbers from 2 up to the entered number. Recall that a number is prime if it is not divisible by any numbers other than 1 and itself.

Ans. This is a nice exercise in building an (inefficient) Pthreads program. The main function is the master thread, and it creates a number of slave threads. Each slave locks a mutex and takes the next un-examined number to work on to see if it is prime. When done, the slave marks this as prime and takes a new number to test. If all numbers have been taken, it exits. The master waits for all slaves to complete by executing a join.

5.9 Make your solution to the previous exercise more efficient. When a slave thread marks a number n as prime, it can mark $2 * n, 3 * n, 5 * n$, etc, as not prime. Other optimisations can be added as well.

Ans. This is a natural extension of the previous problem, but with this exercise, race conditions can be an issue. It is important to ensure that threads are in the critical region for as little time as possible!

5.10 Write a simple FTP server and client using the TCP sockets library on Linux. The client should provide a simple command-line interface where a host and port are provided as arguments. Similarly, the server should provide a simple command line taking a port as argument. Its basic functionality is to allocate a socket and then repeatedly execute the following: to wait for the next connection from a client, to send a short acknowledgement to the client, close the connection, and go back.

Ans. This is a classic problem, and the reader would be advised to start by looking at relevant networking books such as [13].

Chapter 6: Protocols

6.1 Invent a protocol that allows client software to list and buy items from an online shop.

Ans. First, think about the interfaces of the client and server code (i.e., what services are provided by the server, and what needs to be provided by the client). Then, think about the information that is needed by both client and server to support these services. This should help you define the message format. Then focus on identifying a sequence of communication and messages that are needed to support the services. Sometimes, drawing a UML sequence diagram, or even a use case diagram, can help with identifying interfaces and services.

6.2 Recall the alternating bit protocol in Section 6.7. Add an assertion to the Promela specification that states that messages sent cannot be deleted or reordered. Check the property using SPIN.

Ans. Holzmann's book on SPIN [30] has an example of how to do this. A key problem is to ensure that there actually is progress, i.e., that there are no infinite cycles that avoid progress states. Holzmann shows how to use SPIN to check for such cycles.

6.3 The following program, taken from [51], solves the mutual exclusion problem for two processes.

```
boolean flag_1 = false;
boolean flag_2 = false;
enum TURNS { 1, 2 } turn;

/* Define this function for i=1,2 and ensure
   that j=3-i */

void P_i() {
  while(1) {
    NC_i: skip;
     flag_i = true;
    turn = i;
    while(flag_j && turn != j) {
      skip;
    }
    CS_i: skip;
     flag_i = false;
  }
}
```

Describe this program in Promela.

Ans. A plausible solution is in [30]. Here is a sample.

```
bool turn, flag [2];
byte ncrit;

active [2] proctype user()
{
   assert(_pid ==0 || _pid==1);

again:
   flag[_pid] = 1;
   turn = pid;
   (flag[1−_pid]==0 || turn == 1−_pid);
   ncrit++;            /* Critical section */

progress:              /* Progress for next question */
   assert(ncrit == 1);
   ncrit−−;
   flag[_pid] = 0;
   goto again
}
```

6.4 Given your answer to the previous question, using SPIN to validate the mutual exclusion property using assertions. Show that both processes P_1 and P_2 cannot be in their critical sections at the same time.

Ans. See the answer to the previous question, which contains assertions used to validate mutual exclusion.

6.5 **Challenging.** Use SPIN to validate a *progress* property, particularly that either of the processes P_1 and P_2 can enter its critical region over and over again.

Ans. The answer to Exercise 6.3 included progress labels, which can be used to validate this property. Interestingly, we don't need any stronger properties, like weak fairness, because the protocol is implicitly fair — it gives preference to the process that has not just entered the critical region.

6.6 (Adapted from [32].) A water storage system has sensors, a user, and inlet and outlet devices. The sensors measure the water level within a storage device. The outlet device provides water for the user. At each moment, the user decides randomly whether or not to request water. When the water level reaches 20 units, the sensors close the outlet and open the inlet. This causes the water level to rise. When the level reaches 30 units, the inlet is closed and the outlet opened again. The initial water level is 25 units.

Model the water storage system using distinct Promela processes to capture sensors, user, inlet, and outlet. Add an assertion to ensure that the water level is always within the range of 20 to 30 units. Explore the model using the SPIN simulator and verifier.

Ans. A sample solution is

```
mtype = { open, close }

mtype out=open, in=close;
byte  water_level=25;
byte  user_water=0;        /* user  reservoir  */

active  proctype Sensors()
{
  do
  ::  atomic{ (water_level<=20) -> out=close; in=open }
  ::  atomic{ (water_level>=30) -> out=open; in=close }
  od
}

active  proctype User()
{
  do
  ::  (user_water>0) -> user_water--
  ::  true              -> skip
  od
}

active  proctype Inlet ()
{
  do
  ::  (in==open) -> water_level++
  od
}

active  proctype Outlet()
{
  do
  ::  (out==open) -> atomic{ water_level--; user_water++ }
  od
}

active  proctype Monitor()
{
  do
  ::  assert (  water_level>=20 && water_level<=30 )
  od
}
```

6.7 Recall the discussion on the SMTP protocol in Section 6.6 (page 113). Telnet in to an SMTP server (**Warning:** make sure that you have permission to do this!) and work through a similar script as the one presented in Section 6.6, i.e., after connecting, run through the HELO, MAIL FROM, RCPT TO, DATA, and QUIT parts of the protocol. Make a log of your session and indicate in your log where handshaking takes place, and what the server responses mean. What do you think will happen if, instead of typing your own address in MAIL FROM, you typed someone else's address? Make sure that you are the recipient of the email in this case!

Ans. This is mainly to reinforce the SMTP protocol and its lack of authentication!

6.8 Phil wants to send an email to Rich via SMTP. His email client is configured to use the SMTP server `smtp.pracdistprog.com`. In order to connect with the SMTP server, the server's name has to be resolved to an IP address using the domain name service (DNS). What messages will be sent in this process? Assume that only the name server responsible for the domain pracdistprog.com is aware of the requested IP address.

Ans. This question reinforces both the SMTP protocol and lookup from earlier chapters. It is straightforward. Basically, the messages exchanged are:

1. Phil sends a message to the local name server (NS), request for `smtp.pracdistprog.com`.
2. The local NS sends a message to the root NS with this request. The local NS does not know anything about the requested name but it knows the IP address of a root NS, so it forwards the request.
3. The root NS sends a message to the .com NS with this request. The root NS doesn't know the requested name but can pass the request to the NS responsible for the .com domain.
4. The root NS sends a request to the `pracdistprog.com` NS. This NS searches its database and the response is sent back along the same path.

6.9 The *Routing Information Protocol (RIP)* helps routers dynamically adapt. It is used to communicate information about the networks that are reachable from a router, and the distance to those networks. RIP is effectively obsolete and has been subsumed by protocols like OSPF. Research RIP and provide a concise, precise description of it using a suitable language.

Ans. The basic idea is that RIP is a distance-vector routing algorithm. Think of the network as a graph consisting of nodes. In RIP, each node maintains a set of triples (destination, cost, next hop). It exchanges updates with directly connected neighbours (e.g., when tables change). Updates are lists of pairs (destination and cost), and a local table is updated if a better route is received (e.g., lower cost).

6.10 Explain how the routing information protocol from the previous question deals with loops in the network graph, and with failures.

Ans. Loops can be dealt with by setting a finite maximum value for distance, or by using a 'split horizon' approach, i.e., don't send routes learned from a neighbour back to that neighbour. Failures can be dealt with similarly; in some cases a shortest-path approach may need to be applied to deal with failed links.

Chapter 7: Security

7.1 What is the difference between a security policy and a security mechanism?

Ans. A security policy expresses the controls and limits on information in a system. A security mechanism is used to support or implement a policy.

7.2 What security policies does your organisation or institution use for *physical* security?

Ans. This is an interesting research exercise. It is relatively easy to find out policies regarding IT; for example, that a department will secure assets (e.g., machines) against loss by theft, fraud, malicious or accidental damage, or even breach of confidence. Another typical policy is to protect the organisation from liability resulting from facility use. Other aspects of a policy may cover things like building construction (locks, windows), protection from intruders (keycard access, alarms, CCTV), emergency preparedness, reliable power supplies and climate control.

7.3 Suppose that you received an email purporting to come from the IT security group for your organisation. The email claims that the IT group is auditing the key cards used in the organisation (i.e., cards used to open doors). The email requests your key card number and where the card can be used (e.g., your office, the print room). What would you do?

Ans. This is an authorisation problem — do you know the sender of the email, and do you know that the person the sender is claiming to be is authorised to collect this information? (Note that the sender and the real person may be different people!) Before hitting reply you should check that the Reply-To address is really that for whom you are expecting the message to go to (e.g., the head of the IT group). Always check who your messages are going to as they can be redirected. If you don't know this person, you should obtain independent confirmation that the request is legitimate.

7.4 The Bell-LaPadula security model requires that processes must not read data at a higher level ('no read up'), nor can a process write data to a lower level. What is the effect of these two restrictions?

Ans. The first restriction is obvious: a process cannot read data that is more secret than the process is allowed to read. The second restriction is to prevent leaks of data. Specifically, data cannot be declassified in this model. As remarked in Chapter 7, actually enforcing this is difficult, given the possible existence of coding errors and covert channels.

7.5 How does GnuPG protect secret keys?

Ans. Briefly, the secret parts of these keys are protected by secret key encryption. The key for this algorithm is based on the passphrase that you are asked to enter. For most uses of GnuPG, you really should set a passphrase. Additionally, applications like GnuPG will try to prevent the operating system paging out memory containing secrets to prevent the secrets being left written to a swap file or partition. Even then, other applications such as editors might leave interesting material on parts of the computer's disk.

7.6 A *man-in-the-middle* attack involves a third party inserting, changing or reading messages between two other parties without their knowledge. What defences can you think of that could protect against this attack?

Ans. Most defences use authentication techniques based on, e.g., asymmetric cryptography, passwords, strong mutual authentication, or even voice recognition or biometrics. This is a serious security problem, even for quantum cryptography. It is a general problem resulting from the existence of intermediaries acting as proxy for clients: if the proxies are trustworthy (and correct) things should be fine, but if they are not, then vulnerabilities can be targeted.

7.7 A *certificate authority* is an entity (e.g., an organisation) that issues public key certificates. When might such an organisation be useful?

Ans. A certificate authority might be useful if one entity wants to reliably obtain the public key of another — that is, the authority is trusted to provide accurate, reliable information.

7.8 Suppose that Alice receives an email that is apparently digitally signed by Bob. Bob denies ever having sent the email in the first place. Bob's public key is widely available on many key servers. Can it be proven, beyond reasonable doubt (i.e., the criminal standard of proof), that Bob sent the email? Explain.

Ans. This is hard to answer because we have only incomplete information. It is certainly possible for the colleague to have sent the email, but we do not know if the colleague's email client was inaccessible to attackers, whether his private key was obtained by coercion, etc. These possibilities should be investigated, e.g., by systems administrators. The interested reader might wish to examine the Judicial Studies Board's Digital Signature Guidelines [34].

7.9 Music copyright holders are particularly interested in preventing unauthorised digital distribution of music. What mechanisms are used to prevent unauthorised distribution? How effective do you think each of these will be, both in the short term and in the long term?

Ans. Some important mechanisms to bring to light in an answer include digital licenses or watermarking, proprietary encoding formats, and of course legal recourse. Purely technical means are unlikely to be effective.

7.10 Consider the SSL partial example in Section 7.6.5 (page 134). Complete the client implementation. In particular, implement the HTTP request, read the response and provide any necessary error handling. It would also be useful to destroy any objects at the end of the command loop.

Ans. This is not particularly difficult, and the reader is encouraged to look at [49] while writing the implementation. Effectively, the implementation needs to make use of OpenSSL function calls, but much of the effort is in the error handling.

7.11 What are the assumptions associated with the Needham-Schroeder protocol? What can go wrong in the protocol? How might those problems be fixed?

Ans. The problem is that the ticket found by A might be used even after K_A is changed. So an attacker can save information until K_A is known, and then decrypt to find the ticket $T = K_B\{K_{AB}, A\}$. In other words, an attacker can impersonate A. One solution to this problem is the Ottway-Reese protocol, which prevents this problem by encrypting $K_A\{K_{AB}, N_A\}$ with new keys.

Chapter 8: Languages and Distributed Processing

8.1 Examine the producer-consumer example on page 148. Reimplement the program with two or three producers and one consumer.

$\boxed{\mathcal{EG}}$
pc_ex.adb

Ans. The rough approach is to replace the producer task by a task type. Then create the producers as instances of that task type.

8.2 Reimplement the Ada producer-consumer example again, this time with multiple producers and multiple consumers. Each producer should be prepared to have its output handled by any consumer.

$\boxed{\mathcal{EG}}$
pc_ex2.adb

Ans. Again, we use task types. This time, we also need a shared buffer, implemented as a protected object to mediate between the producers and consumers.

8.3 Explain the general steps required to implement a distributed system using Java RMI. How do these steps differ from an implementation using C?

Ans. The steps are: implement a server, expose the services offered by the server in the RMI registry and implement one or more clients. The clients should determine the location of the services via the RMI registry. The registry must then be started. This contrasts with C by providing a higher-level interface for communication — the means by which a service is found and a method invoked is hidden by RMI in Java, whereas the details of the remote procedure call, message wrapping, etc., are visible and accessible to the programmer when using C.

8.4 Explain the purpose of the lookup method of class Naming when using Java RMI.

Ans. This method is used to help clients obtain remote references to objects, via the RMI registry.

8.5 Why does Java not provide *safe* typing?

Ans. Java allows you to dereference null pointers or access arrays out of bounds. There are debugging tools to help with detecting these conditions, like ESC/Java2.

8.6 Assess the suitability of dynamically typed languages like Ruby for building distributed systems.

Ans. Ruby actually provides a distributed programming library which has similarities to Java RMI and also JavaSpaces. Ruby probably falls short in terms of performance for some kinds of distributed systems, as it is interpreted. A good answer should consider whether different kinds of exceptions can occur with dynamic typing, and whether dealing with these in a distributed setting may prove more difficult than in the sequential case.

8.7 Ada programs can use tasks; they can also access fork() via the package Interfaces.C. How do these interact?

Ans. One might expect badly, although it depends on how the Ada tasks are implemented by the run-time system. The reader is encouraged to experiment (and then avoid the idea thereafter).

8.8 Select another programming language that you know, which was not discussed in this book. Compare the language against the criteria that we used in this chapter, and as a result assess its usefulness for building distributed systems.

Ans. This is a research question and students are advised to consult both web references as well as standard documents and/or classic references for programming languages. It is particularly interesting to consider a very different language for building distributed systems, e.g., Python or OCaml.

8.9 Investigate the Eiffel THREAD class available at `http://docs.eiffel.com/`, and contrast it with the Java Thread class at `http://java.sun.com/`. Compare the two classes at both the API level and in terms of how clients might use the classes.

Ans. These classes are very similar in terms of their APIs, modulo differences between the programming languages themselves; moreover, Eiffel includes better executable documentation in terms of contracts. They are also used by clients similarly, but in Eiffel, multiple inheritance is permitted and thus you often see the THREAD class inherited with others (e.g., COMPARABLE).

8.10 Write a SCOOP program that allows shared access to a scoreboard. There are six players and a coordinator (judge), each of which is a process. One individual (player or coordinator) can access the scoreboard at a time. To gain access, a game is played. Each individual guesses a real value between 1 and 10. The players send their values to the coordinator. The players with guesses lower than the coordinator's can play in the next turn, while the other players lose a turn. The player with a guess closest to the coordinator's gets to add data (e.g., their name) to the scoreboard.

Ans. This is a generally straightforward exercise, but it may not be entirely well suited to SCOOP! An interesting alternative to this exercise is to implement it using Ada.

Chapter 9: Building Distributed Systems

9.1 The example PHP implementation of AUTH SMTP in Section 9.3 does not log the responses from the SMTP server. Add logging of all server responses in a sensible way.

Ans. This is straightforward. After each fgets () statement in the PHP code, a response from the server is stored in variable $smtpresponse. This can be easily added to a log array, e.g.,

```
$log['from'] = "$smtpresponse"
```

The only real issue here is how to index the log array, and it is probably easiest to do this using keywords from the SMTP protocol itself, for ease of later lookup.

9.2 Extend the PHP implementation in Section 9.3 to include robust error checking.

Ans. This is very similar to the previous exercise. After adding logging, we should check the SMTP response against a list of standard SMTP error codes; these are documented precisely in [36]. Of course, not all error codes are relevant to each part of the program, but a sensible way to proceed would be to add an array containing all error codes that can be used for lookup. Then, specific error handling can apply after each lookup.

9.3 Research the architecture of a fully fledged email client, such as Mozilla Thunderbird or Pine, or a webmail application such as Horde or SquirrelMail. Discuss how the architecture of this client or application extends the simple architecture we discussed in Section 9.3 (page 163).

Ans. Information on Thunderbird's architecture can be found through the Mozilla documentation. Thunderbird is quite complex as it supports a plug-in architecture, like Firefox. Horde is somewhat more straightforward, and can be studied starting from its API reference at `horde.org`. An interesting advanced classroom discussion would be to contrast a webmail architecture with a plug-in architecture like Thunderbird.

9.4 The Apache JAMES project provides a number of Java solutions for mail and Usenet news. Investigate the structure of the Apache JAMES server, which makes use of SMTP and POP3. In particular, how are the POP3 server and SMTP server related, and how does this server store mail messages and mailboxes?

Ans. Substantial information on JAMES can be found at `james.apache.org`. The Wiki in particular has detailed information that can help in answering this question.

9.5 Secure Copy (SCP) is used to securely transfer files between hosts. It uses the SSH protocol. Research the SCP protocol and clarify both how it uses SSH, and what new aspects it introduces. In particular, clarify how SCP attempts to prevent extraction of useful information from its transmissions.

Ans. The best place to start investigating SCP is one of the sites that provides SCP implementations, e.g., OpenSSH, PuTTY, etc. SCP encrypts data during transfer, but uses SSH directly to provide authentication. SCP works by connecting to a host via SSH, and then executing an SCP server. The client asks for a specific set of files to be uploaded; for downloads, the client sends a list as well, and the server provides the client with additional attributes. Note that because downloads are server-driven, there are additional risks with this.

9.6 For a distributed revision control system like Darcs, draw an architecture diagram similar to the one we presented for Unison in Section 9.5 (page 174).

Ans. Darcs is based on patches, rather than commits and changes; as such, its architecture is rather different. The best place to start researching this is the Darcs wiki: `darcs.net/DarcsWiki`.

9.7 Some distributed revision control systems work by storing the *full* text for the latest revision, and deltas (i.e., changes) for older revisions. Why do you think this is done?

Ans. The general principle here is that the latest revision is the one that will be checked out and manipulated the most; earlier revisions are usually checked out less frequently. To minimise the time taken by the checkout process, these revision control systems support so-called backwards deltas, which allow any previous version to be reached from the baseline (recent) revision. Most RCSs also support forward deltas, for branching, and some older systems have forward merged deltas (e.g., SCCS) which have problems.

Chapter 10: A Networked Game

10.1 In Section 10.3 (page 185), we say that it makes more sense for the tiles handled by a particular map server to be distributed across the world rather than to be adjacent. Why?

Ans. We speculate that the action in such games tends to be concentrated in a few particular areas, rather than spread evenly across the entire map. Thus if a map server handles adjacent areas, it might be heavily loaded, whereas the other map servers handling distant tiles might have no work to do at all.

10.2 Modify the game so that a client can immediately ask for locations of all players and shots.

Ans. A request packet needs to be added, sent from the client to the map server. On receipt, the map server should send a series of UDP packets to the requesting client with the required information. An interesting problem is identifying those players and shots that are no longer in view.

10.3 Modify the game so that the various clients and servers can request updates when UDP messages are missed. How can clients and servers realise they need an update (e.g., for lost messages?).

Ans. Initially, this is easy: a client or server can send a packet, say, REFRESH, which should cause a full update to be sent. But because this might require recursion, e.g., a map server asking its neighbours, we must be careful to ensure that loops don't occur. Some form of unique identifier is useful here in the request packet.

The second part of the question could exploit heartbeat messages that indicate how many packets have been sent, or some other form of sequence numbering.

10.4 Change the game to use SSL for its TCP connections.

Ans. The fragment of SSL code (see Section 7.6.5 (page 134)) is used to replace the TCP connections in the game.

10.5 Extend the clients and admin server to find each other automatically, rather than explicitly giving the admin server's address to the client.

Ans. A broadcast message is sent (e.g., to IP address 255.255.255.255 or a subnet broadcast address): hopefully, a server will receive it and reply. A useful comparison is the DHCP protocol.

10.6 Modify the admin and map servers so that the tiles can be assigned according to the capabilities (e.g., system load) of the individual map servers.

Ans. At the time the map server registers with the admin server, it needs to supply a measure of how many tiles it can accept. assign_tiles () can then be modified to use this measure appropriately. Changing the tile allocation during the game is dealt with in the next question.

10.7 **Challenging.** Change the game so that the tiles can be propagated from one map server to another.

Ans. There are severals aspects to this problem:

1. When and why should a tile propagate? Perhaps when a map server has too high a load, or will be shut down soon (if map servers could be closed down individually). Or perhaps when further map servers become available during the game.
2. Suppose map server A is to propogate tile T to map server B. The admin server needs to update its record. Then the neighbouring map servers need to be updated (via a HANDLER message). The connected clients also need updating. Then A must pass all its state to B. Finally, all this must be done quickly enough so that A can stop, pass its details to B and B can start running; otherwise some form of update-while-running is required.

10.8 Modify the admin server so that players can have a persistent profile.

Ans. This requires that the player has some way to identify themselves on return (typically via password). There are also persistent data, and the map servers need to save any persistent data via the admin server on, for example, shutdown.

10.9 Add a cryptographic structure to the game so that packets can be protected and their source verified.

Ans. The difficult part here is arranging the initial keying. The various servers need to trust each other. Then the initial connection from players needs a key associating. There are substantial performance questions, as well as a more general risk assessment that may make the need for this questionable.

10.10 The game currently allows the admin server to quit in between accepting map server registrations and assigning tiles to them. So map servers can be left hanging for a terminated admin server. How can this be rectified?

Ans. Modify wait_for_tiles () in the map server so that each waits for a UDP or TCP message from the admin server telling it to quit, as well as waiting for its tile allocation.

10.11 Dead players can quit — what effect does this have and why? How can any problems be fixed?

Ans. Dead players aren't currently registered with a map server — so the REMOVE PLAYER message isn't sent to the admin server by a map server. This is a specific instance of the more general problem of player connections just disappearing. Timeouts are needed: if a player hasn't been heard from for a period, then remove them completely.

10.12 Allow the players to have multiple shots in-game at any one time.

Ans. This requires the protocol to be modified so that we can identify each shot uniquely. The relevant data structure and messages require augmenting with this identifier.

10.13 The admin server is responsible for choosing the start location of players, and the current version simply chooses a random location. We have suggested that for fairness, the new player should drop a 'reasonable distance' away from any other player. How could this be arranged?

Ans. The difficulty in this exercise relates to the distributed state. Only the map servers know definitively where each player currently stands. So the admin server has no obvious way to set the location itself. It could choose a map server and expect the map server to locate the player in a 'good' place — this is possible given the state that the map server holds.

10.14 Create additional map servers and clients in other languages, e.g., an Ada map server and a Java client.

Ans. This addresses the fundamental question of interoperability. You may find that the specification given in Chapter 10 is not detailed enough, so you will need to examine the C source for some fine detail.

10.15 Extend the challenge-response protocol between the admin server and map server so that the map server has confidence that it is talking to the right admin server.

Ans. Once the map server has sent its response to the admin server's challenge, it should send its own challenge with a different nonce. Thus it can use exactly the same method that the map server uses.

10.16 The game in this chapter tiles the game world using squares. Consider instead using *hexagons*. What are the advantages and disadvantages of this? Implement such a change.

Ans. The main disadvantage is more complication: in allocation of map servers, handover, etc. The benefit may be an improved space partitioning algorithm, which reduces the overhead needed to manage tile borders. A particularly useful improvement is to use P2P communication at tile borders. An undergraduate project run by the second author of this book [18] explored this architecture, and demonstrable improvements were found.

10.17 Challenging. Split the user login service from the admin server and allow it to be replicated/distributed.

Ans. The login service needs some form of synchronised storage so that two different clients cannot acquire the same login name at the same time via different login servers. Additionally, the login server needs some way to pass authenticated messages to the admin and/or map servers.

B
About the Example Code

The example code is supplied online at `http://www.scm.tees.ac.uk/p.j.brooke/dpb/`. They are supplied with a makefile suitable for GNU make. Tar (including compressed variants) and ZIP archives are available.

The code has been tested primarily on an Intel-based x86 computer running Debian GNU/Linux 3.1 and should work on most Linux or Unix systems. Some, but not all, examples will run using Cygwin.

All C examples can be compiled by typing `make cexamples`.

Similarly, the Ada examples can be compiled by `make adaexamples`. These should work on any platform that supports GNAT using its tasking configuration (at the time of writing, this excludes Cygwin, which does have GNAT but without tasking).

One of the Ada examples, built by `make cspsimexamples`, requires CSPsim.[1] Before running `make`, add a symbolic link to the CSPsim directory. Similarly, the CSPsim lib directory needs to be in `LD_LIBRARY_PATH` before running the resulting executable.

The example `deadlock.csp` is intended for use in FSEL's FDR2 and ProBE tools[2] [22].

[1] `http://www.scm.tees.ac.uk/p.j.brooke/cspsim/`
[2] `http://www.fsel.com/software.html`

Bibliography

[1] J. Abrial. *The B-Book*. Cambridge, 1996.

[2] R. Anderson. *Security Engineering*. Wiley, 2001.

[3] K. Beck and C. Andres. *Extreme Programming Explained*. Addison-Wesley, 2nd edition, 2004.

[4] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations, 1973. `http://www.albany.edu/acc/courses/ia/classics/belllapadula1.pdf`.

[5] D. Bernstein. Using maildir format, 1995. `http://cr.yp.to/proto/maildir.html`.

[6] P. Brooke. CSPsim. `http://www.scm.tees.ac.uk/p.j.brooke/cspsim/`.

[7] P. Brooke, R. Paige, and J. Jacob. A CSP model of Eiffel's SCOOP. To appear in Formal Aspects of Computing, 2007.

[8] A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages*. Addison-Wesley, 1990.

[9] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[10] M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna. Proc. workshop on rigorous engineering of fault tolerant systems, 2005. `http://www.cs.ncl.ac.uk/research/pubs/trs/papers/915.pdf`.

[11] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. RFC 2440: OpenPGP Message Format, 1998. `http://tools.ietf.org/html/rfc2440`.

[12] J. Calvez. *Embedded Real-Time Systems*. Wiley, 1993.

[13] D. Comer. *Computer Networks and Internets*. Prentice-Hall, 4th edition, 2004.

[14] M. Compton. SCOOP: an investigation of concurrency in eiffel. Master's thesis, Australian National University, 2002.

[15] M. Crispin. RFC 3501: Internet Message Access Protocol - Version 4rev1, 2003. `http://tools.ietf.org/html/rfc3501`.

[16] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22, 1976.

[17] E. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5), 1968.

[18] X. Dong. A hybrid network architecture for massively multiplayer online games, B.Eng thesis, 2005. University of York.

[19] ECMA-367. Eiffel analysis, design, and programming language, 2005.

[20] T. Erl. *Service-Oriented Architecture*. Prentice-Hall, 2005.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, 1999. `http://tools.ietf.org/html/rfc2616`.

[22] Formal Systems (Europe) Ltd. Failures-divergence refinement, October 1997. `http://www.formal.demon.co.uk/`.

[23] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.

[24] O. Fuks, J. Ostroff, and R. Paige. SECG: The SCOOP to Eiffel code generator. *Journal of Object Technology*, 3(10), 2004.

[25] D. Gelernter. Generative communication in LINDA. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[26] The Globus Toolkit 4.0.4, 2007. `http://www.globus.org/toolkit/`.

[27] M. Harman. The current state and future of search based software engineering. In *Proc. ICSE 2007: Future of Software Engineering*. ACM Press, 2007.

[28] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 2001.

[29] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International UK, 1985.

[30] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[31] Reference Model for Open Distributed Programming, ISO/IEC CD 10746-1, 1992. International Standards Organisation (ISO).

[32] A. Ireland. Automated Reasoning for Software Engineering Module, 2003. `http://www.macs.hw.ac.uk/~air/ar/mc.html`.

[33] C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, April 2006.

[34] Digital signature guidelines, July 2000. Judicial Studies Board. `http://www.jsboard.co.uk/publications/digisigs/`, last accessed 25 February 2007.

[35] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[36] J. Klensin. RFC 2821: Simple Mail Transfer Protocol, 2001. `http://tools.ietf.org/html/rfc2821`.

[37] K. Kopper. *The Linux Enterprise Cluster*. No Starch Press, 2005.

[38] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[39] S. Maguire. *Writing Solid Code*. Microsoft Press, 1993.

[40] Q. Mahmoud. *Distributed Programming with Java*. Manning, 1999.

[41] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[42] D. Mills. Network Time Protocol (Version 3): Specification, Implementation and Analysis, 1992. `http://tools.ietf.org/html/rfc1305`.

[43] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[44] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[45] J. Myers. RFC 1939: Post Office Protocol - Version 3, 1996. `http://tools.ietf.org/html/rfc1939`.

[46] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Comm. ACM*, 21:993–9, 1978.

[47] P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zürich, 2007.

[48] P. Nienaltowski. SCOOPLI prototype implementation, 2007. `http://se.inf.ethz.ch/research/scoop.html`.

[49] Open SSL 0.9.8e Library, 2007. Open SSL Project. `http://www.openssl.org/`.

[50] T. Ozsu and P. Valduriez. *Principles of Distributed Databases*. Prentice-Hall, 2nd edition, 1999.

[51] G. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3), 1981.

[52] G. Pfister. *In Search of Clusters*. Prentice-Hall, 2nd edition, 1997.

[53] B. Pierce and D. Turner. *Pict: A Programming Language Based on the Pi-Calculus, in Proof Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2001.

[54] B. Pierce and J. Vouillon. What's in Unison? A Formal Specification and Reference Implementation of a File Synchroniser, 2004. `http://www.cis.upenn.edu/~bcpierce`.

[55] J. Postel. RFC 793: Transmission Control Protocol, 1981. `http://tools.ietf.org/html/rfc793`.

[56] S. Rabin, editor. *Introduction to Game Development*. Charles River Media, 2005.

[57] Portland Pattern Repository. Model View Controller, 2006. `http://c2.com/cgi/wiki?ModelViewController`.

[58] P. Resnick. RFC 2822: Internet Message Format, 2001. `http://tools.ietf.org/html/rfc2822`.

[59] R. Rivest. Personal web page, 2007. `http://theory.lcs.mit.edu/~rivest/`.

[60] B. Roscoe. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice Hall, 1998.

[61] S. Schneider. *Concurrent and Real-time Systems*. Wiley, 2000.

[62] B. Schneier. *Beyond Fear*. Copernicus Books, 2003.

[63] R. Seacord. *Secure Coding in C and C++*. Addison-Wesley, 2005.

[64] R. Shirey. RFC 2828: Internet Security Glossary, 2000. `http://www.ietf.org/rfc/rfc2828.txt`.

[65] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 7th edition, 2005.

[66] D. Sklar and A. Trachtenberg. *PHP Cookbook*. O'Reilly, 2006.

[67] R. Slade. *Dictionary of Information Security*. Syngress Media, 2006.

[68] N. Smart. *Cryptography: An Introduction*. McGraw-Hill, 2003.

[69] J. Smith. *Practical OCaml*. APress, 2006.

[70] I. Somerville. *Software Engineering (8th Edition)*. Addison-Wesley, 2006.

[71] M. Spivey. *The Z Reference Manual*. Prentice-Hall, 2001.

[72] T. Sterling, D. Becker, J. Dorband, D. Savarese, U. Ranawake, and C. Packeret. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995. `http://citeseer.ist.psu.edu/sterling95beowulf.html`.

[73] T. Taft and R. Duff, editors. *Ada 95 Reference Manual*, volume 1246 of *Lectures Notes in Computer Science*. Springer-Verlag, 1997.

[74] T. Taft and R. Duff. Ada: Annex e - distributed systems, 1997. `www.adaic.org/standards/95lrm/html/RM-E.html`.

[75] A. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1994.

[76] A. Tanenbaum. *Computer Networks*. Prentice-Hall, 4th edition, 2003.

[77] S. Tardieu. Adasockets. `http://www.rfc1149.net/devel/adasockets`.

[78] A. Tridgell. rsync algorithm, 2006. `http://rsync.samba.org/`.

[79] Ubiquitous Computing Grand Challenge, 2007. `www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/index.html`.

[80] K. Walden and J. Nerson. *Seamless Object-Oriented Software Architecture*. Prentice-Hall, 1994.

[81] Wikipedia. SABRE computer system, 2006. `http://en.wikipedia.org/wiki/Sabre_computer_system`.

[82] Wikipedia. Comparison of programming languages, 2007. `http://en.wikipedia.org/wiki/Comparison_of_programming_languages`.

[83] P. Wojchiechowski and P. Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *Symposium on Agent Systems and Applications*. IEEE, 1999. `http://doi.ieeecomputersociety.org/10.1109/ASAMA.1999.805388`.

[84] T. Ylonen. RFC 4251: The Secure Shell Protocol Architecture, 2006. `http://tools.ietf.org/html/rfc4251`.

[85] T. Ylonen. RFC 4252: The Secure Shell User Authentication Layer Protocol, 2006. http://tools.ietf.org/html/rfc4252.

[86] T. Ylonen. RFC 4254: The Secure Shell Connection Layer Protocol, 2006. http://tools.ietf.org/html/rfc4254.

Glossary

access control	Limiting access to computing services, data and other resources to authorised processes. Access control mechanisms include authentication, authorisation and audit.
accountability	Holding individuals responsible for their actions and their effects on a system.
API	Application programming interface.
architecture	The structures that make up a system, including the system's components, their relationships and their visible characteristics.
asynchronous	Not synchronous; events that are not coordinated, e.g., by a shared clock.
atomic (instruction)	An indivisible operation.
authentication	The process of verifying the identity of an individual or process.
authorisation	The process of granting permission to access resources to individuals or processes.
automaton	A finite state machine, which consists of a set of states and transitions between states, possibly labelled with events or guards indicating when transitions can be taken.
availability	Effectively, the amount of time in which a system is operating within its specification. Also the property where a system is in a suitable state for its clients when needed by them.
available	See *availability*.

avionics	The electronic systems used within an aircraft, e.g., flight control system, propulsion control system.
bag	A set where each element in the set has a multiplicity, indicating the number of occurrences of that element in the set. Also known as a *multiset*.
BitTorrent	A peer-to-peer protocol for file sharing, particularly aimed at providing redundancy.
blocking	A call that does not return until a result or error can be returned.
Byzantine	Usually refers to *byzantine fault tolerance*, where a system must be able to handle a component that behaves arbitrarily, or behaves inconsistently when interacting with other components.
CA	Common acronym for *certificate authority*.
CCS	Milner's Calculus of Communicating Systems [43, 44].
certificate authority	An entity that issues digital certificates, used by other entities; it is an example of a trusted third party.
client-server	A network architecture that separates a client (which may exist in many instances) from a server; the clients send requests to the server, which responds appropriately.
commission	A type of fault that arises when an entity generates incorrect results; a Byzantine fault is a special kind of commission fault, where incorrect results are generated maliciously.
concurrent	Concurrent computations may overlap in time. (Compare with sequential.)
concurrent server	A server capable of providing services to more than one client at the same time.
confidentiality	The process of ensuring that information is accessible to only those authorised with access.
connection-oriented	A form of data transmission, where entities about to exchange data establish a persistent channel for the subsequent messages.

connectionless	A form of data transmission, where each transmitted packet contains sufficient information to allow the packet to be delivered without the aid of additional instructions. Such packets are generally called datagrams.
contention	A conflict over access to resources. For example, lock contention arises when a process attempts to obtain a lock held by another process.
cookie	Data sent from a web server and stored on a client computer. Cookies are typically used to identify users and provide tracking capability.
CORBA	Common Object Request Broker Architecture, a standard for components and communication among them. Supports the composition of heterogeneous components that may be distributed across a network.
crash	A form of fault where a system stops providing its expected function, and generally fails to respond to new instructions.
critical section	A section of code where only one thread of control should be executing at any time.
cryptography	The practice of understanding how to communicate in the presence of adversaries (a definition due to Rivest).
CSP	Hoare's Communicating Sequential Processes [29].
CSPsim	A simulator for CSP [6].
datagram	A packet in a connectionless protocol.
deadlock	A state where two or more processes are waiting for the others to complete their computations — thus, none is able to make progress.
dependable	The trustworthiness of a system; includes aspects of availability, reliability, safety and security.
digital signature	A type of asymmetric cryptography that mimics the security properties of a signature (in digital, as opposed to written form). Intended to detect accidental or deliberate alteration of the signed data.

distributed	Processes or resources that are communicating directly or indirectly across a network. They may or may not be geographically distributed, i.e., in different locations.
DNS	Domain Name System, a distributed Internet database that stores information about Internet addresses.
Domain Name System	See *DNS*.
embedded	A dedicated computer system, typically built into a larger system (e.g., the engine controller of an aircraft) [12].
error	An action performed that is against the specification of a system.
extensibility	The capability to add new features, functionality or resources to a system.
extra-functional	Refers to requirements that provide criteria for judging a system's operation, as opposed to directives regarding its behaviour. Also known as *non-functional*.
failure	A system fails if it does not achieve its objectives; computer systems often fail as a result of faults.
fat client	See *thick client*.
fault	An abnormal condition or defect in a component.
fault tolerant	A system that continues to operate in the presence of faults, instead of terminating or crashing.
FDR2	Failures-Divergence Refinement [22]; a model checker for CSP.
FIFO	First-in, first-out (a queue).
File Transfer Protocol	An application layer protocol for transferring files. This protocol utilises TCP.
FTP	See *File Transfer Protocol*.
full duplex	Simultaneous two-way communication.

Grid	An architecture that considers all resources as manageable entities with common interfaces. Particular focus is on achieving substantial performance.
half duplex	Communication where the signal only ever travels one way. Some authorities swap the meaning for half duplex with simplex; also compare with full duplex.
hard deadline	A deadline that must not be missed, e.g., in a system where an operation not completed by the deadline could cause a failure.
heterogeneous	A heterogeneous system consists of components and parts of diverse nature, e.g., written in different languages, running on different operating systems and different hardware.
HTTP	Hypertext Transfer Protocol, used to transfer information over the WWW; originally used to transfer HTML pages.
HTTPS	Indicates a secure HTTP connection (HTTP over SSL), for providing authenticated and encrypted communication.
IMA	See *integrated modular avionics.*
IMAP	Internet Message Access Protocol, allowing clients to access email on a remote server.
integrated modular avionics	Real-time networked airborne systems consisting of modules capable of supporting many applications at different levels of criticality. Supports reconfiguration of software at run-time in the presence of failure.
integrity	The process of ensuring that data is correct and complete, e.g., after an operation has been applied to it or after the data has been stored or transmitted.
Internet Protocol	The network layer protocol of the TCP/IP family.
IP	See *Internet Protocol*.
IPC	Interprocess communications.
iterative server	A server that handles one client at a time; compare with concurrent server.

Java RMI	Java's facility for identifying and invoking methods on objects that may be remotely located, i.e., executing on a machine other than the one executing the client.
JNI	The Java Native Interface (JNI) allows code running in a Java virtual machine to invoke (and be invoked by) native code written in another language, e.g., C.
Kerberos	An authentication protocol, which prevents eavesdropping and replay attacks.
kernel	The core of an operating system, usually running in supervisor mode.
key	Information that controls a cryptographic algorithm, e.g., the transformation of plain text into encrypted text (or vice versa).
layered protocols	A set of protocols structured in layers, where protocols at one layer use services provided by protocols at a different layer.
loose coupling	A design approach where interfaces of components in a system are developed with few assumptions about how other components are to operate. Loose coupling is generally a desirable design characteristic, as it can improve modularity and extensibility.
loosely-coupled	See *loose coupling*.
MAC	Acronym for *message authentication code.*
maildir	A format for storing email.
maintainable	A maintainable system maximises the ease by which new functionality can be added and existing functionality changed.
man page	Manual pages documenting instructions and commands in Unix-like systems.
manual pages (Unix)	See *man page.*
mbox	A family of file formats for holding email messages.
message	A piece of information used to verify the integrity of a message.

authentication code	
model checker	A tool that implements an algorithm to verify properties of a formal model.
monitor	A programming concept to support synchronisation between two or more tasks, typically using a shared resource.
multitasking	The appearance of simultaneous execution of processes sharing common resources.
mutex	Mutual exclusion.
mutual exclusion	Mechanisms and algorithms to ensure that only one thread of control is allowed in a critical section at any particular time.
name server	A mechanism for recording and accessing information related to domain names, e.g., for translating hostnames into IP addresses.
ncurses	'New curses', a "terminal-independent method of updating character screens".
non-blocking	A call that returns quickly even if there is no result to return.
nonce	A number used once. Typically used in cryptographic protocols.
object-oriented	A design and programming paradigm in which systems are constructed from classes, and in which objects are created at run-time to engage in computations. Paradigm characteristics include polymorphism and inheritance.
ODP	The Open Distributed Processing reference model, which defines the basic concepts and constructs for information management of any system, without considering how the information should be managed.
omission	An omission fault causes a process to not send a message.
OO	See *object-oriented*.
operating system	Software that enables use of the computer hardware, often involving a range of sharing and protection services.
OSI	The Open Systems Interconnection Basic Reference Model, a layered description for network protocol design.

P2P	See *peer-to-peer*.
parallelism	Simultaneous (concurrent) execution of a task or process on multiple processors; generally the task or process must be adapted to support this.
peer	The name given to a host at the far end of a network connection.
peer-to-peer	A network in which all nodes simultaneously can act as clients and servers; connections are typically ad hoc.
PHP	PHP: Hypertext Preprocessor, a programming language designed for dynamic web pages.
polling	Repeatedly checking the value of, say, a variable or device.
POP3	Post Office Protocol, used to retrieve email from a remote server over a TCP/IP connection.
primitive	Generally referring to basic instructions or components that make up larger, more complex structures (e.g., primitive data types, machine code).
process	A process is a running instance of a program.
Promela	Process Meta Language, used to describe systems for analysis with the SPIN model checker.
protected object	An Ada object that provides synchronisation based on a data object (instead of a thread of control). Related to a monitor.
protocol	A set of rules for communication. A protocol describes the format of messages being exchanged as well as the order in which they are sent and received.
quantum	An indivisible entity. A slice of time allocated by a scheduler.
race condition	A critical dependency on the relative timing of events.
real-time	A system for which correctness depends not only on the correctness of a result but also on the time that the result is produced.

reliable	The ability of a system to perform its functions sufficiently well under stated conditions.
remote procedure call	A protocol that allows a program running on one computer to call a subroutine (function, procedure, method) running on a second computer.
replication	The use of multiple instances of a resource, generally to improve reliability and fault tolerance.
RFC	Request for Comments — an Internet standards document.
risk	Generally, the potential negative impact to an asset that has some value to stakeholders. Risk can be defined as the product of the probability of a loss and the value of the loss; this is the *expected loss*.
RMI	Remote Method Invocation, e.g., the ability to invoke a method on an object located on a remote machine.
RPC	Remote Procedure Call, similar to RMI, but generally not involving object-oriented languages.
rsync	A Unix program that synchronises files and directories across a network.
safety	Being protected against the consequences of failure, loss, error or accident.
security	Concerned with managing the risks associated with using a computer.
security mechanism	Means for enforcing or checking a security policy.
security policy	A statement of what is allowed in a system. Describes the aims of the security mechanisms.
semaphore	A mechanism used for enforcing mutual exclusion.
sequential	Ordered and accessed according to that order; for example, sequential access to data, sequential programs. Non-overlapping computations.
service oriented architecture	See *SOA*.

simplex	Communication where the signal travels only one way at a time. Some authorities swap the meaning of simplex with half duplex; also compare with full duplex.
SMTP	Simple Mail Transfer Protocol [36], an important standard for sending email across the Internet.
SOA	Service-Oriented Architecture, an overloaded term which generally refers to a systems architecture made up of loosely coupled (potentially autonomous) services.
SOAP	A standard W3C protocol for exchanging XML messages using HTTP.
socket	An abstraction of a network connection or interface.
software interrupt	Instructions that cause a software context switch to an interrupt handler, e.g., to support multitasking by allowing a scheduler to operate, or to allow a program to access operating system facilities via a system call.
SSH	A set of protocols for producing a secure channel between a local and remote computer, through use of authentication mechanisms based on public-key cryptography and encryption of messages.
SSL	Secure Sockets Layer, a cryptographic protocol for secure Internet-based communications.
starvation	The perpetual denial of resources to a process, preventing the process from completing its work.
stdin	Standard input, usually the keyboard, but can be redirected from a file or pipe.
stream	Typically a sequence of data elements appearing over time. In the C programming language, a stream is an abstraction used for communicating over network sockets, or for reading and writing to files.

synchronisation	Ensuring that access to shared resources occurs at appropriate times. A problem that requires coordination of events or processes so that a task can be completed, with steps occurring in the correct order, and with race conditions avoided.
system call	A request for services from the operating system.
TCP	See *Transmission Control Protocol*.
TCP/IP	A family of network protocols used on Internet-connected computers.
thick client	A client program that handles both presentation of data and business logic.
thin client	A client program that only handles presentation of data; business logic is handled at the server.
thread	A thread is similar to a process, and generally executes like a process. However threads are usually created in a different (often more lightweight) way to processes. They often share an address space with sibling threads.
threat	Anything that has potential to cause harm to a system by exploiting a vulnerability.
threat model	A description of threats to a system, e.g., potential attacks, taking into account likelihood, potential harm and priority.
three-tier	A model comprising clients, application (business logic) servers and database servers.
time-sharing	Sharing resources amongst multiple users by multitasking.
time-slice	The duration in which a process is allowed to execute in a preemptive multitasking system.
Transmission Control Protocol	A reliable, connection-oriented transport protocol of the TCP/IP family.
transparency	The ability of a distributed system to hide elements of its distributed nature from its users.
trap	Another name for a *software interrupt*.
trusted third party	An entity that facilitates communication between two different entities, both of whom trust the TTP.

TTP	See *Trusted Third Party*.
two-tier	A client-server model.
UDDI	Universal Description, Discovery, and Integration.
UDP	See *User Datagram Protocol*.
UML	Unified Modelling Language, a de facto standard modelling language for describing software and systems.
Unison	A file synchroniser.
URL	Uniform Resource Locator, an 'address' to identify and locate resources. Most recently defined in RFC3986.
use case	A description of a sequence of events that generate something useful from a system.
Usenet	An Internet discussion system comprising many newsgroups, in which users can post and read articles.
User Datagram Protocol	An unreliable, connectionless transport protocol of the TCP/IP family.
V model	A graphical representation of the system development life cycle. The life cycle is represented as a V. The left side of the V is where specifications (of requirements, designs, etc) are defined, and the right side is where testing of systems against specifications take place. The bottom of the V represents development.
vulnerability	A weakness in a system, which may allow an attacker to violate the system's security policy.
W3C	World Wide Web consortium. Develops standards and guidelines.
WSDL	Web Services Description Language, an XML-based language for describing web services and their communications.
WWW	World Wide Web, often shorted to Web.

XML	Extensible Markup Language (XML). A general-purpose markup language, increasingly used to format messages in distributed systems.
XML-RPC	A form of remote procedure call that uses XML to encode calls; HTTP is used as the transport mechanism.

Index