

In reality, clients and servers implementing a timeout-and-retransmit policy need to be smarter than this. When our client times out waiting for the next data block from the server, it does not know whether it was the previous outgoing acknowledge packet which got lost, or the new incoming data packet. Examples of the two scenarios are shown in Figure 4.22.

In the first case, the acknowledgement for block 2 is lost and retransmitted. The server sees nothing unusual. In the second case, the data for block 2 are lost. The client times out and retransmits the acknowledgement for block 1. The server will see two copies of this acknowledgement. It must be smart enough to realize this, and to respond by resending data block 2. When the client receives this block, it knows that the server has transmitted it at least once, and possibly twice. Transmitting a file block a second time does no harm – the operation is said to be *idempotent* – that is, it has no harmful effect if repeated. In this case, our concern is to ensure that the operation has happened at least once – if it has happened more than once, no harm is done.

Not all operations are idempotent. Consider a server responsible for maintaining your bank account. A request comes in to transfer £250 to the local tax office. The server performs the operation, and sends an acknowledgement, which gets lost. The client, tiring of waiting for a reply, times out and resends the request. Clearly, the server must *not* repeat the transfer of funds. It must be smart enough to realize ‘I’ve already done that’, and simply resend the acknowledgement. Typically, this is achieved by including some form of unique *transaction identifier* with each request. The server, maintaining a cache of recently processed transaction identifiers, along with the result of that transaction, is able to reject the duplicate request, return the cached result, and so preserve a guarantee of having performed the operation ‘exactly once’ even in the face of lost or duplicated packets.

## 4.8 A concurrent, distributed application

---

We will conclude this chapter with an example of a true distributed, concurrent application. The problem we are going to tackle can be stated very simply: we want to know how many prime numbers there are between 1 and 1 000 000. It is, of course, conceivable that you do not wish to know this. Nonetheless, the example has some merit; finding prime numbers is a relatively time-consuming business, and the idea is to get several servers working concurrently on different parts of the problem, to reduce the overall time to reach a solution. This is a clear-cut example of *data distribution*, discussed back in Section 1.3.1.

### 4.8.1 A nondistributed program to count prime numbers

To begin, here is an ordinary, nondistributed program to do the job:

```

#define SMALLEST      1L
#define BIGGEST      1000000L

main(argc, argv)
int argc;
char *argv[];
{
    long count;
    count = count_primes(SMALLEST, BIGGEST);
    printf("Between %ld and %ld there are %ld primes\n",
          SMALLEST, BIGGEST, count);
}

long count_primes(min, max)
long min, max;
{
    long i, count = 0;
    for (i=min; i<=max; i++)
        if (isprime(i)) count++;
    return count;
}

int isprime(n)
long n;
{
    int i;
    for (i=2; i*i <= n; i++)
        if (n%i == 0)
            return 0;
    return 1;
}

```

This program takes 277 seconds to execute on a Sparcstation 2, and 935 seconds on a Sun 3/50. Clearly, with run-times this long, a concurrent distributed solution would be useful. (The answer, by the way, is 78 499.)

## 4.8.2 A server to count prime numbers

Before writing a distributed version of this program, there are two key design decisions to be taken. The first is: how do we divide the work between the client and the (multiple) servers? The second is: should the servers be connection-oriented or connectionless?

There are two possible places to draw the line between server and client. The first is to implement just the `isprime()` function on the server. That is to say, we send the server a number, and it sends back a boolean telling if the number is prime or not. This is clearly an appalling choice; the amount of work the server needs to do on each transaction is much too small. The communications overhead would far exceed the actual computation time, and we would end up with a distributed version which ran *slower* than the original

single-processor version. The second choice is to implement `count_primes()` on the server. On each transaction we send the server two numbers specifying the lower and upper limits of the number range to be searched, and the server returns the count of primes within that range. Providing the range is fairly large, the computational load dominates and the communication overhead becomes negligible. The client, then, simply needs to divide up the total range into a number of subranges in some way, pass each subrange over to a server, collect the replies from the servers, and add them up. This is the basic model we will implement. The idea is to start up servers on multiple machines, so that they may operate concurrently.

The second decision, about whether to implement a connection-oriented or connectionless server, is much less clear-cut. It could be satisfactorily done either way. The fact that each transaction with the server is self-contained and makes no reference to previous transactions possibly argues for a connectionless service. Also, it is a little easier for the client to retrieve replies from multiple servers if they are all sent to the same UDP socket, rather than to separate TCP connections. On the other hand, use of UDP may force us to add in a timeout and retransmit mechanism to recover from lost packets if the subnetwork is not reliable. Choosing a suitable timeout value would be difficult because there is no easy way to estimate how long a particular request ought to take. It is easier to have the timeout and retransmission handled ‘underneath’ the application code. Consequently, we will choose to implement a connection-oriented server, using TCP.

As just discussed, the client-to-server protocol will consist simply of a pair of numbers specifying the subrange. In reply, the server just sends a single value back to the client, indicating the number of primes in that range.

With these design decisions out of the way, it is a simple matter to write the server. Indeed, as we have already seen the boiler-plate code for a TCP server, and we already have the `count_primes()` function, there is very little new to write.

There are a couple of things about which the client and server need to agree – which port number to use, and the format of the request packet. We will place these shared definitions in a header file, `primes.h`:

```
/* primes.h */
#define PRIME_PORT 1066
struct subrange {
    long      min;
    long      max;
};
```

Now here is the complete server:

```
/* Server for counting prime numbers */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
```

```

#include "primes.h"

main()
{
    int sock, msgsock, client_len;
    struct sockaddr_in server, client;
    long count;
    struct limits limits;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        exit(1);
    }

    server.sin_family      = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port        = htons(PRIME_PORT);

    if (bind(sock, (struct sockaddr *) &server, sizeof (server)) < 0) {
        exit(2);
    }

    listen(sock, 5);
    while (1) {
        client_len = sizeof(client);
        msgsock = accept(sock, (struct sockaddr *) &client, &client_len);
        if (msgsock < 0) {
            exit(3);
        }

        /* Once a connection is made, we accept multiple requests from
           the client, until the client drops the connection. At that
           point, we get an error back from the read()
        */

        while (1) {
            if(read(msgsock, &limits, sizeof limits) != sizeof limits)
                /* Assume 'end of file'; drop the connection */
                break;
            count = count_primes(limits.min, limits.max);
            write(msgsock, &count, sizeof count);
        }
        close(msgsock);
    } /* Go wait for the next client */
}

/* The count_primes() and isprime() functions go here. They are the same
   as in the previous example.
*/

```

### 4.8.3 A simple prime number counting client

The client is a little harder. How does it know on which machines the server is available? How does it divide up the range? How does it keep track of which server is doing what? How does it collect the answers? How does it know when they are all in?

We will make a few simplifying assumptions to keep our client as easy as possible. Later, we will develop a more sophisticated version. To begin with, we assume that the servers have all been started up manually, for instance by logging in to each remote workstation in turn. Therefore, we know precisely where the servers are running, and can supply the appropriate machine names to the client on the command line. The range subdivision will be accomplished simply by dividing the 1–1 000 000 range into a number of equal sized subranges, depending on how many servers we were given. We will open a TCP connection to each server and send it a subrange. Then we will read back (in the same order) the responses from the servers, and total them.

---

**An aside:** our choice of a binary protocol implicitly assumes that the machines running the client and the server are using compatible binary data representations. We will examine this problem further – and present a solution to it – in Chapter 6.

---

Here is our initial, simple-minded client:

```

/* Client for counting prime numbers: Version 1
   C.R.Brown  24 June 1992
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include "primes.h"

#define MAX_HOSTS      10
#define SMALLEST      1
#define BIGGEST        1000000

main(int argc, char *argv[])
{
    int connection[MAX_HOSTS];
    char *host[MAX_HOSTS];
    int i, nhosts;
    long count, total;
    struct subrange limits;
    long start_time;

    nhosts = argc - 1;

```

```

if (nhosts < 1) {
    fprintf(stderr, "%s: no hosts specified\n", argv[0]);
    exit(1);
}

if (nhosts > MAX_HOSTS) {
    nhosts = MAX_HOSTS;
    fprintf(stderr, "Too many hosts, using only the first %d\n", nhosts);
}

/* Try to obtain connections to each server */
for (i=0; i<nhosts; i++) {
    host[i] = argv[i+1];
    switch (connection[i] = connect_to_server(host[i], PRIME_PORT)) {
        case -1: fprintf(stderr, "Problem creating socket - bye!\n");
                exit(2);
        case -2: fprintf(stderr, "Unknown host: %s - bye!\n", host[i]);
                exit(3);
        case -3: fprintf(stderr, "Cannot find server on host %s - bye!\n",
                        host[i]);
                exit(4);
        default: printf("connected to host %s\n", host[i]);
    }
}

/* Send a subrange to each server */
start_time = time(0);
limits.max = SMALLEST - 1;
for (i=0; i<nhosts; i++) {
    limits.min = limits.max + 1;
    limits.max = limits.min + (BIGGEST - SMALLEST + 1)/nhosts;
    if (i == nhosts-1) limits.max = BIGGEST;
    printf("sending range (%d,%d) to host %s\n",
           limits.min, limits.max, host[i]);
    write(connection[i], &limits, sizeof limits);
}

/* Read responses back from servers, in same order */
total = 0;
for (i=0; i<nhosts; i++) {
    read(connection[i], &count, sizeof count);
    printf("got reply = %ld from host %10s after %ld sec\n",
           count, host[i], time(0)-start_time);
    total += count;
}
printf("answer is %ld\n", total);
}

int connect_to_server(char *host, int port)
{

```

```

int sock;
struct sockaddr_in server;
struct hostent *host_info;

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;

if ((host_info = gethostbyname(host)) == NULL) return -2;

server.sin_family = AF_INET;
server.sin_port = htons(port);
memcpy(&server.sin_addr, host_info->h_addr, host_info->h_length);

if (connect(sock, &server, sizeof server) < 0) return -3;

return sock;
}

```

Now suppose we have compiled and somehow started the server on the remote hosts `douglas`, `benjie` and `frankie`. We could then run our client, telling it to use these three hosts, as follows:

```

eddie% prime_client douglas frankie benjie
connected to host douglas
connected to host frankie
connected to host benjie
sending range (1,333334) to host douglas
sending range (333335,666668) to host frankie
sending range (666669,1000000) to host benjie
got reply = 28666 from host douglas after 151 sec
got reply = 25405 from host frankie after 163 sec
got reply = 24428 from host benjie after 404 sec
answer is 78499
eddie%

```

Whilst our client clearly works, it has some major limitations. For one thing, it does not react well to the situation in which some of the servers cannot be contacted. It simply gives up completely, and exits. It would be better simply to ignore the servers we cannot find, and carry on with the rest. This requires a little additional logic, which we will incorporate in the next version.

#### 4.8.4 Load-balancing issues

A much more interesting problem with the client becomes apparent if you examine the timings on the output shown above. Hosts `douglas` and `frankie` take approximately the same time to complete their work, but `benjie` takes nearly three times as long. Why is this? It is mainly because `benjie` is dealing with larger numbers and it takes longer, on average, to test a large number for prime-ness than to test a small one. The timings are also affected, of course, by the speed of the individual processors. Whatever the reasons,

we are clearly not exploiting the machines as well as we might, because douglas's and frankie's servers are both spending about 60 per cent of their time idle. In the jargon of the parallel processing fraternity, we say that the load is *unbalanced*.

How can we fix this? One possible solution is to split the (1–1000000) range into unequal-sized pieces, such that each piece presents roughly the same computational load. To do this, we need some way of estimating the computational load of a given subrange. Unfortunately, prime numbers being what they are, there is no easy way to do this. (This is not always the case with this type of application. Sometimes, there is a cheap way to estimate the effort to process a given server request, which can be used to perform 'up front' load-balancing.)

Similar difficulties with a priori load-balancing arise in other, less trivial applications. For example, a distributed image processing application used spatial parallelism, dividing the image into horizontal strips, in order to speed up a stereo matching algorithm. The execution time of this algorithm was strongly dependent on the amount of edge detail in the image, so that dividing the image into equal-sized strips did not generally balance the load very well. The problem, as in our prime number example, is that there is no cheap way of estimating the amount of work involved in processing a given subset of the data.

Even if such a cost metric were available, it would not take into account the effect of different processor speeds, which are, usually, not known to the client. Furthermore, the timings will be affected by how heavily the host machines are loaded by other, unrelated tasks.

#### 4.8.5 Implementing a processor farm

To fix this problem we will adopt a particular type of client-server paradigm known in parallel computing circles as a *processor farm*. The idea is as follows: a single client (called a master, in processor farm jargon) farms out the work to multiple servers (or slaves). It divides the work up into a large number of 'work packets'. It does not matter if the packets are not all 'equally difficult', but there should be several times as many work packets as there are servers. To begin, it sends a work packet to each server. Then, it waits until any server replies. The reply is recorded, and the next work packet is sent to that server. The process continues, with each server receiving a new work packet as soon as it has finished the last one, until all the packets have been sent out, and all the replies have been received. Notice that the processor farm, with its one client serviced by many servers, is almost the opposite of the usual client-server model, in which one server serves many clients.

The biggest advantage of the processor farm is that it keeps all the servers busy all the time, i.e. it balances the load. There is no need for an up-front estimate of how long each work packet will take. It does not matter if the servers run at different speeds. The biggest disadvantage is that processor farms are only appropriate if the task can be broken into a relatively large number of *independent* work packets. There should be several times as many work packets as there are slave processors. If the servicing of one work

packet depends on results from previous work packets, the client logic gets *much* harder. Fortunately, in our example, each subrange work packet can be processed without reference to any other subrange, making the processor farm a good choice.

**An aside:** in the early days of transputers, the computation of the Mandelbrot set was a favourite amongst vendors as a demonstration of parallel computing. (The Mandelbrot set is a pattern based on fractal geometry which can be explored at arbitrary levels of detail.) There were three reasons for its popularity: it is very computationally intensive, it is very easy to implement as a processor farm, and you get a rather pretty picture at the end of it. Consequently at exhibitions one could walk through the hall seeing screen after screen of Mandelbrot. After a while the market sobered up, Mandelbrot-free zones were declared, and vendors finally started showing development and debugging tools which the users had by then discovered were essential if one was to have any hope of programming anything other than the Mandelbrot set. . . .

There are, however, some important image-rendering applications, such as ray tracing, which can also be parallelized using a processor farm.

No changes are needed to the server to implement the processor farm. On the client side, we now find that our decision to use a connection-oriented server makes life a little difficult for us. The central issue is that the client needs to retrieve responses from the servers in the order in which they occur, which will not in general be the same as the order in which the requests were sent out. If the service was connectionless, the client could collect the responses from all the servers by simply reading from a single datagram socket. In our connection-oriented client, we have a separate file descriptor to read for each server, and of course, we do not know in which order they will become ready.

Earlier in the chapter we saw the use of the `select()` system call to allow a process to block until any one of several specified file descriptors became ready for reading. There, we used it at the server end, to implement a concurrent server without using multiple processes. In our present example, we will use it in the client to concurrently feed data to multiple servers.

Here is our new 'processor farm' version of the client:

```

/* Client for counting prime numbers: Version 2
   Implements a processor farm, using sockets and select()
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include "primes.h"

#define FD_TABLE_SIZE 32

```

```

#define SMALLEST      1
#define BIGGEST      1000000
#define GRANULARITY  100 /* This controls the number of subranges
                          (i.e. work packets) which the total
                          range is divided into */

long next_min = SMALLEST; /* Tracks the lower bound of the next
                          work packet to be sent out */

char *host[FD_TABLE_SIZE];

void send_next_work_packet();
main(int argc, char *argv[])
{
    int i, fd;
    int prospects; /* Prospective number of hosts */
    int outstanding; /* Number of unanswered work packets */
    long count, total;
    long start_time;
    fd_set select_set; /* Set of descriptors to select on */
    fd_set ready_set;

    start_time = time(0);
    prospects = argc - 1;
    if (prospects < 1) {
        fprintf(stderr, "%s: no hosts specified\n", argv[0]);
        exit(1);
    }

    /* Try to obtain connections to each server */
    outstanding = 0;
    argv++; /* Bump pointer past the command name */
    FD_ZERO(&select_set);
    for (i=0; i<prospects; i++) {
        switch (fd = connect_to_server(argv[i], PRIME_PORT)) {
            case -1: fprintf(stderr, "Problem creating socket\n");
                    exit(2);
            case -2: fprintf(stderr, "Unknown host %s: ignored\n",
                            argv[i]);
                    break;
            case -3: fprintf(stderr, "No server on host %s: ignored\n",
                            argv[i]);
                    break;
            default: if (fd >= FD_TABLE_SIZE) {
                    fprintf(stderr, "too many hosts: %s ignored\n",
                            argv[i]);
                    break;
                }
                printf("connected to host %s\n", argv[i]);
                /* It is not essential to record the host name,

```

```

        but it helps us print more meaningful messages
    */
    host[fd] = argv[i];

    /* We do not explicitly record the value of the
       descriptor for this connection, we simply add
       it into the select set. Later, when a select
       returns, we can find out which descriptor is
       ready for reading.
    */

    FD_SET(fd, &select_set);
    send_next_work_packet(fd);
    outstanding++;
}
}

/* At this point, since we have not retrieved any responses,
   the value of outstanding tells us how many hosts we found
*/
if (outstanding == 0) {
    fprintf(stderr, "No servers found!\n");
    exit(3);
}
total = 0;          /* Total count is accumulated in here */

/* Keep the servers busy until the job is done */
while (outstanding > 0) {

    /* We copy select_set onto ready_set because select() over-
       writes its argument, and we need to keep select_set intact
    */

    memcpy(&ready_set, &select_set, sizeof select_set);
    select(FD_TABLE_SIZE, &ready_set, NULL, NULL, NULL);

    /* Now we have to scan through the file descriptor set which
       select() returned to see which is ready for reading. We
       always scan the entire set, because it is possible (though
       unlikely) for several descriptors to become ready at the
       same time.
    */

    for (fd=3; fd < FD_TABLE_SIZE; fd++) {
        if (FD_ISSET(fd, &ready_set)) {
            read(fd, &count, sizeof count);
            printf("got reply = %ld from host %10s after %ld sec\n",
                   count, host[fd], time(0)-start_time);
            total += count;
            outstanding--;

            /* If there are more work packets to process, send

```

```

        the next one out. Otherwise, just mop up the
        outstanding requests
    */
    if (next_min <= BIGGEST) {
        send_next_work_packet(fd);
        outstanding++;
    }
}
}
printf("answer is %ld\n", total);
}

int connect_to_server(char *host, int port)
{
    int sock;
    struct sockaddr_in server;
    struct hostent *host_info;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;
    if ((host_info = gethostbyname(host)) == NULL) return -2;

    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    memcpy(&server.sin_addr, host_info->h_addr, host_info->h_length);

    if (connect(sock, &server, sizeof server) < 0) return -3;

    return sock;
}

void send_next_work_packet(int fd)
{
    struct subrange limits;

    limits.min = next_min;
    limits.max = limits.min + (BIGGEST - SMALLEST)/GRANULARITY;
    if (limits.max > BIGGEST) limits.max = BIGGEST;
    write(fd, &limits, sizeof limits);
    printf("sent range (%d,%d) to host %s\n",
        limits.min, limits.max, host[fd]);
    next_min = limits.max + 1;
}

```

Here is the dialogue when we run it:

```

douglas% prime_farm localhost eddie magrathea desiato benjie
connected to host localhost
sent range (1,10000) to host localhost

```

```

connected to host eddie
sent range (10001,20000) to host eddie
connected to host magrathea
sent range (20001,30000) to host magrathea
connected to host desiato
sent range (30001,40000) to host desiato
No server on host benjie: ignored
got reply = 1230 from host localhost after 1 sec
sent range (40001,50000) to host localhost
got reply = 983 from host magrathea after 2 sec
sent range (50001,60000) to host magrathea
got reply = 958 from host desiato after 2 sec
sent range (60001,70000) to host desiato
got reply = 1033 from host eddie after 3 sec
sent range (70001,80000) to host eddie
got reply = 924 from host magrathea after 4 sec
sent range (80001,90000) to host magrathea

```

... and so on ...

```

got reply = 720 from host magrathea after 138 sec
sent range (980001,990000) to host magrathea
got reply = 711 from host desiato after 140 sec
sent range (990001,1000000) to host desiato
got reply = 710 from host magrathea after 144 sec
got reply = 732 from host localhost after 146 sec
got reply = 721 from host desiato after 146 sec
got reply = 717 from host eddie after 150 sec
answer is 78499

```

The operation of the processor farm is clearly visible from the above dialogue. Although not obvious from what is shown (most of it has been elided out), each host managed to process a different number of work packets, as follows:

localhost	22 packets handled
eddie	16 packets handled
magrathea	31 packets handled
desiato	31 packets handled

Since all the machines were more or less otherwise idle when this test was run, these numbers are indicative of the relative performance of the four machines used.

#### 4.8.6 Starting up the remote servers

We have assumed in the above tests that servers had somehow been started on each machine. This could be done, of course, with an appropriate entry in a boot-time script

(or by starting up the server via `inetd` – both these issues are discussed in Chapter 7). However, let us consider how we might take our client one final step further by having it explicitly startup servers as required on the remote machines.

To make the task a little easier we will make a small modification to the server, so that it automatically creates a child to do the real work. This is easily done by adding the lines

```
if (fork()) exit(0);
close(0);
close(1);
close(2);
```

at the beginning of the `main()` function in the server. The first line causes the parent to exit, leaving the child to get on with the work. The other lines explicitly close the server's `stdin`, `stdout` and `stderr` streams. This step is necessary to ensure that the `rsh` commands shown below do not wait until the child has exited. Of course, closing your `stdout` and `stderr` streams in this way restricts your options for error reporting – any calls to `perror()`, for example, are doomed to failure. We will examine the issue of error logging from background servers in Chapter 7.

This change immediately makes it easier to start up the required servers manually. Rather than logging in on each machine, we can now use a sequence of `rsh` commands on the local machine:

```
eddie% rsh douglas prime_server
eddie% rsh benjie prime_server
eddie% rsh magrathea prime_server
```

We can also bury these operations inside the client. One way is to use the `system()` function:

```
system("rsh douglas prime_server");
system("rsh benjie prime_server");
system("rsh magrathea prime_server");
```

This call causes the specified command to be executed ‘as if it had been typed at a terminal’. This is wonderfully easy in terms of programming effort, but expensive in terms of execution. Each call to `system()` performs a `fork/exec` of a Bourne shell and supplies the argument string to that shell. A `fork()` and an `exec()`, particularly of a big program like a shell, consume significant resources.

A more efficient approach is to use functions such as `rcmd()` or `rexec()`, which we examined in Chapter 3, to contact the remote execution server directly.