20

Distributed Programming

With distributed programming, you can build applications running on several machines that work together through a network to accomplish a task. The computation model described here is parallel programming with distributed memory. Local and remote programs communicate using a *network protocol*. The best-known and most widely-used of these is IP (*Internet protocol*) and its TCP and UDP layers. Beginning with these low-level layers, many services are built on the *client-server* model, where a server waits for requests from different clients, processes those requests, and sends responses. As an example, the HTTP protocol allows communication between Web browsers and Web servers. The distribution of tasks between clients and servers is suitable for many different software architectures.

The Objective Caml language offers, through its `Unix` library, various means of communication between programs. Sockets allow communication through the TCP/IP and UDP/IP protocols. This part of the `Unix` library has been ported to Windows. Because you can create "heavyweight" processes with `Unix.fork` as well as lightweight processes with `Thread.create`, you can create servers that accept many requests at once. Finally, an important point when creating a new service is the definition of a protocol appropriate to the application.

Outline of the Chapter

This chapter presents the basic elements of the Internet, sockets, for the purpose of building distributed applications (particularly client-server applications) while detailing the problems in designing communications protocols.

The first section briefly explains the Internet, its addressing system and its main services.

The second section illustrates communications through sockets between different Objective Caml processes, both local and remote.

The third section describes the client-server model, while presenting server programs and universal clients.

The fourth section shows the importance of communications protocols for building network services.

This chapter is best read after the chapters on systems programming (Chapter 18) and on concurrent programming (Chapter 19).

The Internet

The Internet is a network of networks. Their interconnection is organized as a hierarchy of domains, subdomains, and so on, through *interfaces*. An interface is the hardware in a computer that allows it to be connected (typically, an Ethernet card). Some computers may have several interfaces. Each interface has a unique IP address that respects, in general, the interconnection hierarchy. Message routing is also organized hierarchically: from domain to domain; then from domain to subdomains, and so on, until a message reaches its destination interface. Besides their interface addresses, computers usually also have a name, as do domains and subdomains. Some machines have a particular role in the network:

bridges connect one network to another;

routers use their knowledge of the topology of the Internet to route data;

name servers track the correspondence between machine names and network addresses.

The purpose of the Internet protocol (*i.e.*, of the IP) is to make the network of networks into a single entity. This is why one can speak of **the** Internet. Any two machines connected via the Internet can communicate. Many kinds of machines and systems coexist on the Internet. All of them use IP protocols and most of them, the UDP and TCP layers.

The different protocols and services used by the Internet are described in RFC's (Requests For Comments), which can be found on the Jussieu mirror site:

Link: ftp://ftp.lip6.fr/pub/rfc

Internet Protocols and Services

The unit of transfer used by the IP protocol is the *datagram* or *packet*. This protocol in unreliable: it does not assure proper order, safe arrival, or non-duplication of transmitted packets. It only deals with correct routing of packets and signaling of errors

when a packet is unable to reach its destination. Addresses are coded into 32 bits in the current version of the protocol: IPv4. These 32 bits are divided into four fields, each containing values between 0 and 255. IP addresses are written with the four fields separated by periods, for example: `132.227.60.30`.

The IP protocol is in the midst of an important change made necessary by the exhaustion of address space and the growing complexity of routing problems due to the expansion of the Internet. The new version of the IP protocol is IPv6, which is described in [Hui97].

Above IP, two protocols allow higher-level transmissions: UDP (User Datagram Protocol, and TCP (Transfer Control Protocol). These two protocols use IP for communication between machines, also allowing communication between applications (or programs) running on those machines. They deal with correct transmission of information, independent of contents. The identification of applications on a machine is done via a *port number*.

UDP is a connectionless, unreliable protocol: it is to applications as IP is to interfaces. TCP is a connection-oriented, reliable protocol: it manages acknowledgement, retransmission, and ordering of packets. Further, it is capable of optimizing transmission by a windowing technique.

The standard services (applications) of the Internet most often use the client-server model. The server manages requests by clients, offering them a specific service. There is an asymmetry between client and server. The services establish high-level protocols for keeping track of transmitted contents. Among the standard services, we note:

- FTP (File Transfer Protocol);
- TELNET (Terminal Protocol);
- SMTP (Simple Mail Transfer Protocol);
- HTTP (Hypertext Transfer Protocol).

Other services use the client-server model:

- NFS (Network File System);
- X-Windows
- Unix services: rlogin, rwho ...

Communication between applications takes place via sockets. Sockets allow communication between processes residing on possibly different machines. Different processes can read and write to sockets.

The **`Unix`** *Module and* IP *Addressing*

The `Unix` library defines the abstract type *inet_addr* representing Internet addresses, as well as two conversion functions between an internal representation of addresses and strings:

```
# Unix.inet_addr_of_string ;;
```

```
- : string -> Unix.inet_addr = <fun>
# Unix.string_of_inet_addr ;;
- : Unix.inet_addr -> string = <fun>
```

In applications, Internet addresses and port numbers for services (or service numbers) are often replaced by names. The correspondence between names and address or number is managed using databases. The `Unix` library provides functions to request data from these databases and provides datatypes to allow storage of the obtained information. We briefly describe these functions below.

Address tables. The table of addresses (*hosts database*) contains the assocation between machine name(s) and interface address(es). The structure of entries in the address table is represented by:

```
# type host_entry =
    { h_name : string;
      h_aliases : string array;
      h_addrtype : socket_domain;
      h_addr_list : inet_addr array } ;;
```

The first two fields contain the machine name and its aliases; the third contains the address type (see page 627); the last contains a list of machine addresses.

A machine name is obtained by using the function:

```
# Unix.gethostname ;;
- : unit -> string = <fun>
# let my_name = Unix.gethostname() ;;
val my_name : string = "estephe.inria.fr"
```

The functions that query the address table require an entry, either the name or the machine address.

```
# Unix.gethostbyname ;;
- : string -> Unix.host_entry = <fun>
# Unix.gethostbyaddr ;;
- : Unix.inet_addr -> Unix.host_entry = <fun>
# let my_entry_byname = Unix.gethostbyname my_name ;;
val my_entry_byname : Unix.host_entry =
  {Unix.h_name="estephe.inria.fr"; Unix.h_aliases=[|"estephe"|];
   Unix.h_addrtype=Unix.PF_INET; Unix.h_addr_list=[|<abstr>|]}
# let my_addr = my_entry_byname.Unix.h_addr_list.(0) ;;
val my_addr : Unix.inet_addr = <abstr>

# let my_entry_byaddr = Unix.gethostbyaddr my_addr ;;
val my_entry_byaddr : Unix.host_entry =
  {Unix.h_name="estephe.inria.fr"; Unix.h_aliases=[|"estephe"|];
   Unix.h_addrtype=Unix.PF_INET; Unix.h_addr_list=[|<abstr>|]}

# let my_full_name = my_entry_byaddr.Unix.h_name ;;
val my_full_name : string = "estephe.inria.fr"
```

These functions raise the `Not_found` exception in case the request fails.

Table of services. The table of services contains the correspondence between service names and port numbers. The majority of Internet services are standardized. The structure of entries in the table of services is:

```
# type service_entry =
    { s_name : string;
      s_aliases : string array;
      s_port : int;
      s_proto : string } ;;
```

The first two fields are the service name and its eventual aliases; the third field contains the port number; the last field contains the name of the protocol used.

A service is in fact characterized by its port number and the underlying protocol. The query functions are:

```
# Unix.getservbyname ;;
- : string -> string -> Unix.service_entry = <fun>
# Unix.getservbyport ;;
- : int -> string -> Unix.service_entry = <fun>
# Unix.getservbyport 80 "tcp" ;;
- : Unix.service_entry =
{Unix.s_name="www"; Unix.s_aliases=[|"http"|]; Unix.s_port=80;
 Unix.s_proto="tcp"}
# Unix.getservbyname "ftp" "tcp" ;;
- : Unix.service_entry =
{Unix.s_name="ftp"; Unix.s_aliases=[||]; Unix.s_port=21; Unix.s_proto="tcp"}
```

These functions raise the `Not_found` exception if they cannot find the service requested.

Sockets

We saw in chapters 18 and 19 two ways to perform interprocess communication, namely, pipes and channels. These first two methods use a logical model of concurrency. In general, they do not give better performance to the degree that the communicating processes share resources, in particular, the same processor. The third possibility, which we present in this section, uses sockets for communication. This method originated in the Unix world. Sockets allow communication between processes executing on the same machine or on different machines.

Description and Creation

A socket is responsible for establishing communication with another socket, with the goal of transferring information. We enumerate the different situations that may be encountered as well as the commands and datatypes that are used by TCP/IP sockets. The classic metaphor is to compare sockets to telephone sets.

- In order to work, the machine must be connected to the network (`socket`).
- To receive a call, it is necessary to possess a number of the type *sock_addr* (`bind`).
- During a call, it is possible to receive another call if the configuration allows it (`listen`).

- It is not necessary to have one's own number to call another set, once the connection is established in both directions (`connect`).

Domains. Sockets belong to different *domains*, according to whether they are meant to communicate internally or externally. The `Unix` library defines two possible domains corresponding to the type constructors:

```
# type socket_domain = PF_UNIX | PF_INET;;
```

The first domain corresponds to local communication, and the second, to communication over the Internet. These are the principal domains for *sockets*.

In the following, we use sockets belonging only to the Internet domain.

Types and protocols. Regardless of their domain, sockets define certain communications properties (reliability, ordering, etc.) represented by the type constructors:

```
# type socket_type = SOCK_STREAM | SOCK_DGRAM | SOCK_SEQPACKET | SOCK_RAW ;;
```

According to the type of socket used, the underlying communications protocol obeys definite characteristics. Each type of communication is associated with a default protocol.

In fact, we will only use the first kind of communication — `SOCK_STREAM` — with the default protocol TCP. This guarantees reliability, order, prevents duplication of exchanged messages, and works in connected mode.

For more information, we refer the reader to the Unix literature, for example [Ste92].

Creation. The function to create sockets is:

```
# Unix.socket ;;
- : Unix.socket_domain -> Unix.socket_type -> int -> Unix.file_descr = <fun>
```

The third argument allows specification of the protocol associated with communication. The value 0 is interpreted as "the default protocol" associated with the pair (domain, type) argument used for the creation of the socket. The value returned by this function is a file descriptor. Thus such a value can be used with the standard input-output functions in the `Unix` library.

We can create a TCP/IP socket with:

```
# let s_descr = Unix.socket Unix.PF_INET Unix.SOCK_STREAM 0 ;;
val s_descr : Unix.file_descr = <abstr>
```

Warning

> Even though the `socket` function returns a value of type *file_descr*, the system distinguishes descriptors for a files and those associated with sockets. You can use the file functions in the `Unix` library with descriptors for sockets; but an exception is raised when a classical file descriptor is passed to a function expecting a descriptor for a socket.

Closing. Like all file descriptors, a socket is closed by the function:

```
# Unix.close ;;
- : Unix.file_descr -> unit = <fun>
```

When a process finishes via a call to `exit`, all open file descriptors are closed automatically.

Addresses and Connections

A socket does not have an address when it is created. In order to setup a connection between two sockets, the caller must know the address of the receiver.

The address of a socket (TCP/IP) consists of an IP address and a port number. A socket in the Unix domain consists simply of a file name.

```
# type sockaddr =
    ADDR_UNIX of string | ADDR_INET of inet_addr * int ;;
```

Binding a socket to an address. The first thing to do in order to receive calls after the creation of a socket is to *bind* the socket to an address. This is the job of the function:

```
# Unix.bind ;;
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

In effect, we already have a socket descriptor, but the address that is associated with it at creation is hardly useful, as shown by the following example:

```
# let (addr_in, p_num) =
    match Unix.getsockname s_descr with
      Unix.ADDR_INET (a,n) → (a,n)
    |  _ → failwith "not INET" ;;
val addr_in : Unix.inet_addr = <abstr>
val p_num : int = 0
# Unix.string_of_inet_addr addr_in ;;
- : string = "0.0.0.0"
```

We need to create a useful address and to associate it with our socket. We reuse our local address `my_addr` as described on page 626 and choose port 12345 which, in general, is unused.

```
# Unix.bind s_descr (Unix.ADDR_INET(my_addr, 12345)) ;;
- : unit = ()
```

Listening and accepting connections. It is necessary to use two operations before our socket is completely operational to receive calls: define its listening capacity and allow it to accept connections. Those are the respective roles of the two functions:

```
# Unix.listen ;;
- : Unix.file_descr -> int -> unit = <fun>
# Unix.accept ;;
- : Unix.file_descr -> Unix.file_descr * Unix.sockaddr = <fun>
```

The second argument to the `listen` function gives the maximum number of connections. The call to the `accept` function waits for a connection request. When `accept` finishes, it returns the descriptor for a socket, the so-called *service socket*. This service socket is automatically linked to an address. The `accept` function may only be applied to sockets that have called `listen`, that is, to sockets that have setup a queue of connection requests.

Connection requests. The function reciprocal to `accept` is;

```
# Unix.connect ;;
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

A call to *`Unix.connect s_descr s_addr`* establishes a connection between the local socket *`s_descr`* (which is automatically bound) and the socket with address *`s_addr`* (which must exist).

Communication. From the moment that a connection is established between two sockets, the processes owning them can communicate in both directions. The input-output functions are those in the `Unix` module, described in Chapter 18.

Client-server

Interprocess communication between processes on the same machine or on different machines through TCP/IP sockets is a mode of point-to-point asynchronous communication. The reliability of such transmissions is assured by the TCP protocol. It is nonetheless possible to simulate the broadcast to a group of processes through point-to-point communication to all receivers.

The roles of different processes communicating in an application are asymmetric, as a general rule. That description holds for client-server architectures. A server is a process (or several processes) accepting requests and trying to respond to them. The client, itself a process, sends a request to the server, hoping for a response.

Client-server Action Model

A server provides a *service* on a given port by waiting for connections from future clients. Figure 20.1 shows the sequence of principal tasks for a server and a client.

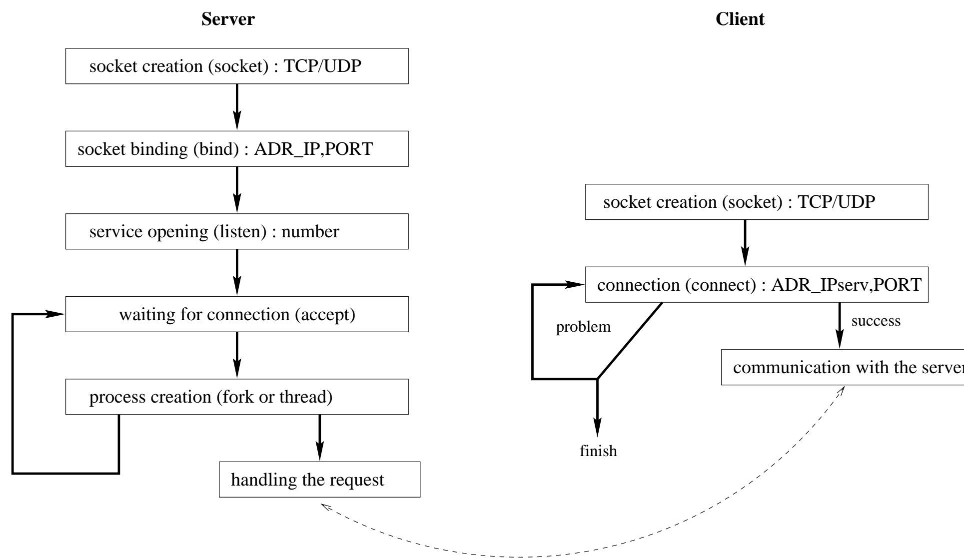

Figure 20.1: Model of a server and client

A client can connect to a service once the server is ready to accept connections (`accept`). In order to make a connection, the client must know the IP number of the server machine and the port number of the service. If the client does not know the IP number, it needs to request name/number resolution using the function `gethostbyname`. Once the connection is accepted by the server, each program can communicate via input-output channels over the sockets created at both ends.

Client-server Programming

The mechanics of client-server programming follows the model described in Figure 20.1. These tasks are always performed. For these tasks, we write generic functions parameterized by particular functions for a given server. As an example of such a program, we describe a server that accepts a connection from a client, waits on a socket until a line of text has been received, converting the line to CAPITALS, and sending back the converted text to the client.

Figure 20.2 shows the communication between the service and different clients[1].

Certain tasks run on the same machine as the server, while others are found on remote machines.

We will see

1. Note of translator: “boulmich” is a colloquial abbreviation for “Boulevard Saint-Michel”, one the principal avenues of Quartier Latin in Paris...

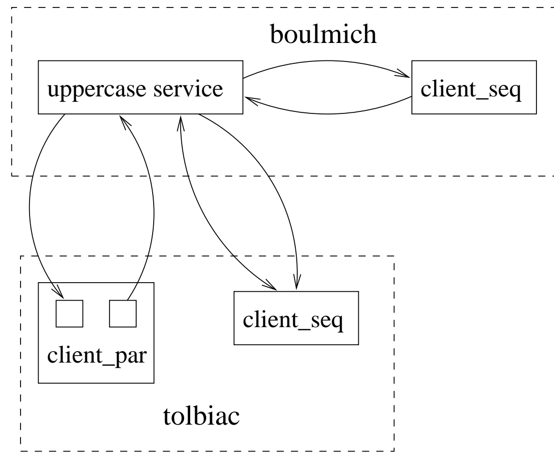

Figure 20.2: CAPITAL service and its clients

1. How to write the code for a “generic server” and instantiate it for our particular capitalization service.
2. How to test the server, without writing the client, by using the `telnet` program.
3. How to create two types of clients:
   - a sequential client, which waits for a response after sending a request;
   - a parallel client, which separates the send and receive tasks.

   Therefore, there are two processes for this client.

Code for the Server

A server may be divided into two parts: waiting for a connection and the following code to handle the connection.

A Generic Server

The generic server function `establish_server` described below takes as its first argument a function for the service (`server_fun`) that handles requests, and as its second argument, the address of the socket in the Internet domain that listens for requests. This function uses the auxiliary function `domain_of`, which extracts the domain of a socket from its address.

In fact, the function `establish_server` is made up of high-level functions from the `Unix` library. This function sets up a connection to a server.

```
# let establish_server server_fun sockaddr =
    let domain = domain_of sockaddr in
    let sock = Unix.socket domain Unix.SOCK_STREAM 0
```

```
    in Unix.bind sock sockaddr ;
       Unix.listen sock 3;
       while true do
         let (s, caller) = Unix.accept sock
         in match Unix.fork() with
                0 → if Unix.fork() <> 0 then exit 0 ;
                     let inchan = Unix.in_channel_of_descr s
                     and outchan = Unix.out_channel_of_descr s
                     in server_fun inchan outchan ;
                        close_in inchan ;
                        close_out outchan ;
                        exit 0
              | id → Unix.close s; ignore(Unix.waitpid [] id)
       done ;;
val establish_server :
  (in_channel -> out_channel -> 'a) -> Unix.sockaddr -> unit = <fun>
```

To finish building a server with a standalone executable that takes a port number parameter, we write a function `main_server` which takes a parameter indicating a service. The function uses the command-line parameter as the port number of a service. The auxiliary function `get_my_addr`, returns the address of the local machine.

```
# let get_my_addr () =
    (Unix.gethostbyname(Unix.gethostname())).Unix.h_addr_list.(0) ;;
val get_my_addr : unit -> Unix.inet_addr = <fun>

# let main_server  serv_fun =
    if Array.length Sys.argv < 2 then Printf.eprintf "usage : serv_up port\n"
    else try
           let port =  int_of_string Sys.argv.(1) in
           let my_address = get_my_addr()
           in establish_server serv_fun  (Unix.ADDR_INET(my_address, port))
         with
           Failure("int_of_string") →
             Printf.eprintf "serv_up : bad port number\n" ;;
val main_server : (in_channel -> out_channel -> 'a) -> unit = <fun>
```

Code for the Service

The general mechanism is now in place. To illustrate how it works, we need to define the service we're interested in. The service here converts strings to upper-case. It waits for a line of text over an input channel, converts it, then writes it on the output channel, flushing the output buffer.

```
# let uppercase_service ic oc =
    try while true do
          let s = input_line ic in
          let r = String.uppercase s
          in output_string oc (r^"\n") ; flush oc
        done
    with _ → Printf.printf "End of text\n" ; flush stdout ; exit 0 ;;
```

```
val uppercase_service : in_channel -> out_channel -> unit = <fun>
```

In order to correctly recover from exceptions raised in the Unix library, we wrap the initial call to the service in an *ad hoc* function from the Unix library:

```
# let go_uppercase_service () =
    Unix.handle_unix_error main_server uppercase_service ;;
val go_uppercase_service : unit -> unit = <fun>
```

Compilation and Testing of the Service

We group the functions in the file serv_up.ml, adding an actual call to the function go_uppercase_service. We compile this file, indicating that the Unix library is linked in

```
ocamlc -i -custom -o serv_up.exe unix.cma serv_up.ml -cclib -lunix
```

The transcript from this compilation (using the option -i) gives:

```
val establish_server :
  (in_channel -> out_channel -> 'a) -> Unix.sockaddr -> unit
val main_server : (in_channel -> out_channel -> 'a) -> unit
val uppercase_service : in_channel -> out_channel -> unit
val go_uppercase_service : unit -> unit
```

We launch the server by writing:

```
serv_up.exe 1400
```

The port chosen here is 1400. Now the machine where the server was launched will accept connections on this port.

Testing with telnet

We can now begin to test the server by using an existing client to send and receive lines of text. The telnet utility, which normally is a client of the telnetd service on port 23, and used to control a remote connection, can be diverted from this role by passing a machine name and a different port number. This utility exists on several operating systems. To test our server under Unix, we type:

```
$ telnet boulmich 1400
Trying 132.227.89.6...
Connected to boulmich.ufr-info-p6.jussieu.fr.
Escape character is '^]'.
```

The IP address for boulmich is 132.227.89.6 and its complete name, which contains its domain name, is boulmich.ufr-info-p6.jussieu.fr. The text displayed by telnet indicates a successful connection to the server. The client waits for us to type on the keyboard, sending the characters to the server that we have launched on boulmich on port 1400. It waits for a response from the server and displays:

```
The little cat is dead.
THE LITTLE CAT IS DEAD.
We obtained the expected result.
WE OBTAINED THE EXPECTED result.
```

The phrases entered by the user are in lower-case and those sent by the server are in upper-case. This is exactly the role of this service, to perform this conversion.

To exit from the client, we need to close the window where it was run, by executing the kill command. This command will close the client's socket, causing the server's socket to close as well. When the server displays the message "End of text," the process associated with the service terminates.

The Client Code

While the server is naturally parallel (we would like to handle a particular request while accepting others, up to some limit), the client may or may not be so, according to the nature of the application. Below we give two versions of the client. Beforehand, we present two functions that will be useful for writing these clients.

The function open_connection from the Unix library allows us to obtain a couple of input-output channels for a socket.

The following code is contained in the language distribution.

```
# let open_connection sockaddr =
    let domain = domain_of sockaddr in
    let sock = Unix.socket domain Unix.SOCK_STREAM 0
    in try Unix.connect sock sockaddr ;
           (Unix.in_channel_of_descr sock , Unix.out_channel_of_descr sock)
      with exn → Unix.close sock ; raise exn ;;
val open_connection : Unix.sockaddr -> in_channel * out_channel = <fun>
```

Similarly, the function shutdown_connection closes down a socket.

```
# let shutdown_connection inchan =
    Unix.shutdown (Unix.descr_of_in_channel inchan) Unix.SHUTDOWN_SEND ;;
val shutdown_connection : in_channel -> unit = <fun>
```

A Sequential Client

From these functions, we can write the main function of a sequential client. This client takes as its argument a function for sending requests and receiving responses.

This function analyzes the command line arguments to obtain connection parameters before actual processing.

```
# let main_client client_fun  =
     if Array.length Sys.argv < 3
     then Printf.printf "usage :  client server port\n"
     else let server = Sys.argv.(1) in
          let server_addr =
            try  Unix.inet_addr_of_string server
            with Failure("inet_addr_of_string") →
                   try  (Unix.gethostbyname server).Unix.h_addr_list.(0)
                   with Not_found →
                          Printf.eprintf "%s : Unknown server\n" server ;
                          exit 2
          in try
               let port = int_of_string (Sys.argv.(2)) in
               let sockaddr = Unix.ADDR_INET(server_addr,port) in
               let ic,oc = open_connection sockaddr
               in client_fun ic oc ;
                  shutdown_connection ic
             with Failure("int_of_string") → Printf.eprintf "bad port number";
                                              exit 2 ;;
val main_client : (in_channel -> out_channel -> 'a) -> unit = <fun>
```

All that is left is to write the function for client processing.

```
# let client_fun ic oc =
     try
       while true do
         print_string  "Request : " ;
         flush stdout ;
         output_string oc ((input_line stdin)^"\n") ;
         flush oc ;
         let r = input_line ic
         in Printf.printf "Response : %s\n\n" r;
            if r = "END" then ( shutdown_connection ic ; raise Exit) ;
       done
     with
         Exit → exit 0
       | exn → shutdown_connection ic ; raise exn  ;;
val client_fun : in_channel -> out_channel -> unit = <fun>
```

The function `client_fun` enters an infinite loop which reads from the keyboard, sends a string to the server, gets back the transformed upper-case string, and displays it. If the string is `"END"`, then the exception `Exit` is raised in order to exit the loop. If another exception is raised, typically if the server has shut down, the function ceases its calculations.

The client program thus becomes:

```
# let go_client () = main_client client_fun ;;
val go_client : unit -> unit = <fun>
```

We place all these functions in a file named `client_seq.ml`, adding a call to the function `go_client`. We compile the file with the following command line:

```
ocamlc -i -custom -o client_seq.exe unix.cma client_seq.ml -cclib -lunix
```

We run the client as follows:

```
$ client_seq.exe boulmich 1400
Request : The little cat is dead.
Response: THE LITTLE CAT IS DEAD.

Request : We obtained the expected result.
Response: WE OBTAINED THE EXPECTED RESULT.

Request : End
Response: END
```

The Parallel Client with `fork`

The parallel client mentioned divides its tasks between two processes: one for sending, and the other for receiving. The processes share the same socket. The functions associated with each of the processes are passed to them as parameters.

Here is the modified program:

```
# let main_client client_parent_fun client_child_fun =
    if Array.length Sys.argv < 3
    then Printf.printf "usage :  client server port\n"
    else
      let server = Sys.argv.(1) in
      let server_addr =
        try  Unix.inet_addr_of_string server
        with Failure("inet_addr_of_string")
               → try  (Unix.gethostbyname server).Unix.h_addr_list.(0)
                  with Not_found →
                         Printf.eprintf "%s : unknown server\n" server ;
                         exit 2
      in try
           let port = int_of_string (Sys.argv.(2)) in
           let sockaddr = Unix.ADDR_INET(server_addr,port) in
           let ic,oc = open_connection sockaddr
           in match Unix.fork () with
                  0 → if Unix.fork() = 0 then client_child_fun oc ;
                        exit 0
                | id → client_parent_fun ic ;
                        shutdown_connection ic ;
                        ignore (Unix.waitpid [] id)
         with
           Failure("int_of_string") → Printf.eprintf "bad port number" ;
                                       exit 2 ;;
```

```
val main_client : (in_channel -> 'a) -> (out_channel -> unit) -> unit = <fun>
```

The expected behavior of the parameters is: the (grand)child sends the request and the parent receives the response.

This architecture has the effect that if the child needs to send several requests, then the parent receives the responses to requests as each is processed. Consider again the preceding example for capitalizing strings, modifying the client side program. The client reads the text from one file, while writing the response to another file. For this we need a function that copies from one channel, `ic`, to another, `oc`, respecting our little protocol (that is, it recognizes the string `"END"`).

```
# let copy_channels ic oc =
    try while true do
          let s = input_line ic
          in if s = "END" then raise End_of_file
             else (output_string oc (s^"\n"); flush oc)
        done
    with End_of_file → () ;;
val copy_channels : in_channel -> out_channel -> unit = <fun>
```

We write the two functions for the child and parent using the parallel client model:

```
# let child_fun in_file out_sock =
    copy_channels in_file out_sock ;
    output_string out_sock ("FIN\n") ;
    flush out_sock ;;
val child_fun : in_channel -> out_channel -> unit = <fun>
# let parent_fun out_file in_sock = copy_channels in_sock out_file ;;
val parent_fun : out_channel -> in_channel -> unit = <fun>
```

Now we can write the main client function. It must collect two extra command line parameters: the names of the input and output files.

```
# let go_client () =
   if Array.length Sys.argv < 5
     then Printf.eprintf "usage : client_par server port filein fileout\n"
   else let in_file = open_in Sys.argv.(3)
        and out_file = open_out Sys.argv.(4)
        in main_client (parent_fun out_file) (child_fun in_file) ;
           close_in in_file ;
           close_out out_file ;;
val go_client : unit -> unit = <fun>
```

We gather all of our material into the file `client_par.ml` (making sure to include a call to `go_client`), and compile it. We create a file `toto.txt` containing the text to be converted:

```
The little cat is dead.
We obtained the expected result.
```

We can test the client by typing:

```
client_par.exe boulmich 1400 toto.txt result.txt
```

The file `result.txt` contains the text:

```
$ more result.txt
THE LITTLE CAT IS DEAD.
WE OBTAINED THE EXPECTED RESULT.
```

When the client finishes, the server always displays the message `"End of text"`.

Client-server Programming with Lightweight Processes

The preceding presentation of code for a generic server and a parallel client created processes via the `fork` primitive in the `Unix` library. This works well under Unix; many Unix services are implemented by this technique. Unfortunately, the same cannot be said for Windows. For portability, it is preferable to write client-server code with lightweight processes, which were presented in Chapter 19. In this case, it becomes necessary to examine the interactions among different server processes.

Threads and the `Unix` *Library*

The simultaneous use of lightweight processes and the `Unix` library causes all active threads to block if a system call does not return immediately. In particular, reads on file descriptors, including those created by `socket`, are blocking.

To avoid this problem, the `ThreadUnix` library reimplements most of the input-output functions from the `Unix` library. The functions defined in that library will only block the thread which is actually making the system call. As a consequence, input and output is handled with the low-level functions `read` and `write` found in the `ThreadUnix` library.

For example, the standard function for reading a string of characters, `input_line`, is redefined in such a way that it does not block other threads while reading a line.

```
# let my_input_line  fd =
    let s = " "  and  r = ref ""
    in while (ThreadUnix.read fd s 0 1 > 0) && s.[0] <> '\n' do r := !r ^s done ;
       !r ;;
val my_input_line : Unix.file_descr -> string = <fun>
```

Classes for a Server with Threads

Now let us recycle the example of the CAPITALIZATION service, this time giving a version using lightweight processes. Shifting to threads poses no problem for our little application on either the server side or the client side, which start processes independently.

Earlier, we built a generic server parameterized over a service function. We were able to achieve this kind of abstraction by relying on the functional aspect of the Objective Caml language. Now we are about to use the object-oriented extensions to the language to show how objects allow us to achieve a comparable abstraction.

The server is organized into two classes: `serv_socket` and `connection`. The first of these handles the service startup, and the second, the service itself. We have introduced some print statements to trace the main stages of the service.

The `serv_socket` class. has two instance variables: `port`, the port number for the service, and `socket`, the socket for listening. When constructing such an object, the initializer opens the service and creates this socket. The `run` method accepts connections and creates a new `connection` object for handling requests. The *serv_socket* uses the *connection* class described in the following paragraph. Usually, this class must be defined before the *serv_socket* class.

```
# class serv_socket p =
    object (self)
      val port = p
      val mutable sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0

      initializer
        let my_address = get_my_addr ()
        in Unix.bind sock (Unix.ADDR_INET(my_address,port)) ;
           Unix.listen sock 3

      method private client_addr = function
          Unix.ADDR_INET(host,_) → Unix.string_of_inet_addr host
        | _ → "Unexpected client"

      method run () =
        while(true) do
          let (sd,sa) = ThreadUnix.accept sock in
          let connection = new connection(sd,sa)
          in Printf.printf "TRACE.serv: new connection from %s\n\n"
                           (self#client_addr sa) ;
          ignore (connection#start ())
        done
    end ;;
class serv_socket :
  int ->
  object
    val port : int
    val mutable sock : Unix.file_descr
    method private client_addr : Unix.sockaddr -> string
    method run : unit -> unit
  end
```

It is possible to refine the server by inheriting from this class and redefining the `run` method.

The `connection` class. The instance variables in this class, `s_descr` and `s_addr`, are initialized to the descriptor and the address of the socket created by `accept`. The methods are `start`, `run`, and `stop`. The `start` creates a thread calling the two other methods, and returns its thread identifier, which can be used by the calling instance of `serv_socket`. The `run` method contains the core functionality of the service. We have slightly modified the termination condition for the service: we exit on receipt of an empty string. The `stop` service just closes the socket descriptor for the service.

Each new connection has an associated number obtained by calling the auxiliary function `gen_num` when the instance is created.

```
# let gen_num = let c = ref 0 in (fun () → incr c; !c) ;;
val gen_num : unit -> int = <fun>
# exception Done ;;
exception Done
# class connection (sd,sa) =
    object (self)
      val s_descr = sd
      val s_addr = sa
      val mutable number = 0
      initializer
        number <- gen_num();
        Printf.printf "TRACE.connection : object %d created\n" number ;
        print_newline()

      method start () =  Thread.create (fun x → self#run x ; self#stop x) ()

      method stop() =
        Printf.printf "TRACE.connection : object finished %d\n" number ;
        print_newline () ;
        Unix.close s_descr

     method run () =
       try
         while true do
           let line =  my_input_line s_descr
           in if (line = "") or (line = "\013") then raise Done ;
              let result = (String.uppercase line)^"\n"
              in ignore (ThreadUnix.write s_descr result 0 (String.length result))
         done
       with
          Done → ()
        | exn  → print_string (Printexc.to_string exn) ; print_newline()
    end ;;
class connection :
  Unix.file_descr * 'a ->
  object
    val mutable number : int
    val s_addr : 'a
    val s_descr : Unix.file_descr
```

```
    method run : unit -> unit
    method start : unit -> Thread.t
    method stop : unit -> unit
  end
```

Here again, by inheritance and redefinition of the `run` method, we can define a new service.

We can test this new version of the server by running the `protect_serv` function.

```
# let go_serv () = let s = new serv_socket 1400 in s#run () ;;
# let protect_serv () = Unix.handle_unix_error go_serv () ;;
```

Multi-tier Client-server Programming

Even though the client-server relation is asymmetric, nothing prevents a server from being the client of another service. In this way, we have a communication hierarchy. A typical client-server application might be the following:

- a mail client presents a friendly user interface;
- a word-processing program is run, followed by an interaction with the user;
- the word-processing program accesses a database.

One of the goals of client-server applications is to alleviate the processing of centralized machines. Figure 20.3 shows two client-server architectures with three tiers.

Each tier may run on a different machine. The user interface runs on the machine running the user mail application. The processing part is handled by a machine shared by a collection of users, which itself sends requests to a remote database server. With this application, a particular piece of data may be sent to the user mail application or to the database server.

Some Remarks on the Client-server Programs

In the preceding sections, we constructed servers for a simple CAPITALIZATION service. Each server used a different approach for its implementation. The first such server used the Unix fork mechanism. Once we built that server, it became possible to test it with the telnet client supplied with the Unix, Windows, and MacOS operating systems. Next, we built a simple first client. We were then able to test the client and server together. Clients may have tasks to manage between communications. For this purpose, we built the `client_par.exe` client, which separates reading from writing by using forks. A new kind of server was built using threads to clearly show the relative independence of the server and the client, and to bring input-output into this setting. This server was organized into two easily-reused classes. We note that both functional programming and object-oriented programming support the separation of "mechanical," reusable code from code for specialized processing.

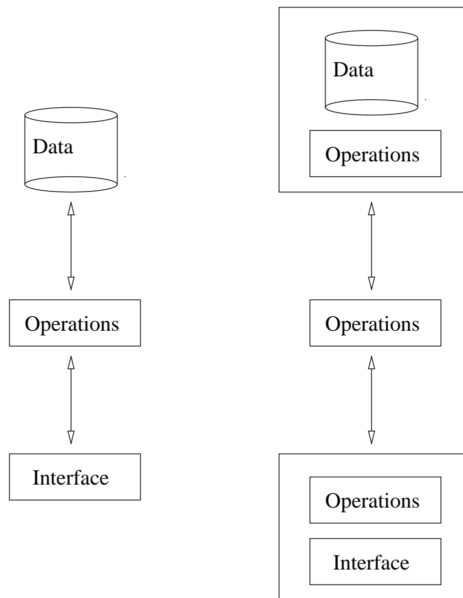

Figure 20.3: Different client-server architectures

Communication Protocols

The various client-server communications described in the previous section consisted of sending a string of characters ending in a carriage-return and receiving another. However simple, this communication pattern defines a protocol. If we wish to communicate more complex values, such as floats, matrices of floats, a tree of arithmetic expressions, a closure, or an object, we introduce the problem of encoding these values. Many solutions exist according to the nature of the communicating programs, which can be characterized by the implementation language, the machine architecture, and in certain cases, the operating system. Depending on the machine architecture, integers can be represented in many different ways (most significant bits on the left, on the right, use of tag bits, and size of a machine word). To communicate a value between different programs, it is necessary to have a common representation of values, referred to as the *external* representation[2]. More structured values, such as records, just as integers, must have an external representation. Nonetheless, there are problems when certain languages allow constructs, such as bit-fields in C, which do not exist in other languages. Passing functional objects or objects, which contain pieces of code, poses a new difficulty. Is the code byte-compatible between the sender and receiver, and does there exist a mechanism for dynamically loading the code? As a general rule, the problem is simplified by supposing that the code exists on both sides. It is not the

2. Such as the XDR representation (eXternal Data Representation), which was designed for C programs.

code itself that is transmitted, but information that allows it to be retrieved. For an object, the instance variables are communicated along with the object's type, which allows retrieval of the object's methods. For a closure, the environment is sent along with the address of its code. This implies that the two communicating programs are actually the same executable.

A second difficulty arises from the complexity of linked exchanges and the necessity of synchronizing communications involving many programs.

We first present text protocols, later discussing acknowledgements and time limits between requests and responses. We also mention the difficulty of communicating internal values, in particular as it relates to interoperability between programs written in different languages.

Text Protocol

Text protocols, that is, communication in ASCII format, are the most common because they are the simplest to implement and the most portable. When a protocol becomes complicated, it may become difficult to implement. In this setting, we define a grammar to describe the communication format. This grammar may be rich, but it will be up to the communicating programs to handle the work of coding and interpreting the text strings sent and received.

As a general rule, a network application does not allow viewing the different layers of protocols in use. This is typified by the case of the HTTP protocol, which allows a browser to communicate with a Web site.

The HTTP Protocol

The term "HTTP" is seen frequently in advertising. It corresponds to the communication protocol used by Web applications. The protocol is completely described on the page of the W3 Consortium:

Link: http://www.w3.org

This protocol is used to send requests from browsers (Communicator, Internet Explorer, Opera, etc.) and to return the contents of requested pages. A request made by a browser contains the name of the protocol (HTTP), the name of the machine (www.ufr-info-p6.jussieu.fr), and the path of the requested page (/Public/Localisation/index.html). Together these components define a URL (Uniform Resource Locator):

```
http://www.ufr-info-p6.jussieu.fr/Public/Localisation/index.html
```

When such a URL is requested by a browser, a connection over a socket is established between the browser and the server running on the indicated server, by default on port 80. Then the browser sends a request in the HTTP format, like the following:

```
GET /index.html HTTP/1.0
```

The server responds in the protocol HTTP, with a header:

```
HTTP/1.1 200 OK
Date: Wed, 14 Jul 1999 22:07:48 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 AuthMySQL/2.20
Last-Modified: Thu, 10 Jun 1999 12:53:46 GMT

Accept-Ranges: bytes
Content-Length: 3663
Connection: close
Content-Type: text/html
```

This header indicates that the request has been accepted (code 200 OK), the kind of server, the modification date for the page, the length of the send page and the type of content which follows. Using the GET commmand in the protocol (HTTP/1.0), only the HTML page is transferred. The following connection with telnet allows us to see what is actually transmitted:

```
$ telnet www.ufr-info-p6.jussieu.fr 80
Trying 132.227.68.44...
Connected to triton.ufr-info-p6.jussieu.fr.
Escape character is '^]'.
GET


<!-- index.html -->
<HTML>
<HEAD>
<TITLE>Serveur de l'UFR d'Informatique de Pierre et Marie Curie</TITLE>
</HEAD>
<BODY>

<IMG SRC="/Icons/upmc.gif" ALT="logo-P6" ALIGN=LEFT HSPACE=30>
Unit&eacute; de Formation et de Recherche 922 - Informatique<BR>
Universit&eacute; Pierre et Marie Curie<BR>
4, place Jussieu<BR>
75252 PARIS Cedex 05, France<BR><P>
....
</BODY>
</HTML>
<!-- index.html -->

Connection closed by foreign host.
```

The connection closes once the page has been copied. The base protocol is in text mode so that the language may be interpreted. Note that images are not transmitted with the page. It is up to the browser, when analyzing the syntax of the HTML page, to

observe anchors and images (see the `IMG` tags in the transmitted page). At this time, the browser sends a new request for each image encountered in the HTML source; there is a new connection for each image. The images are displayed when they are received. For this reason, images are often displayed in parallel.

The HTTP protocol is simple enough, but it transports information in the HTML language, which is more complex.

Protocols with Acknowledgement and Time Limits

When a protocol is complex, it is useful that the receiver of a message indicate to the sender that it has received the message and that it is grammatically correct. The client blocks while waiting for a response before working on its tasks. If the part of the server handling this request has a difficulty interpreting the message, the server must indicate this fact to the client rather than ignoring the request. The HTTP protocol has a system of error codes. A correct request results in the code 200. A badly-formed request or a request for an unauthorized page results in an error code 4xx or 5xx according to the nature of the error. These error codes allow the client to know what to do and allow the server to record the details of such incidents in its log files.

When the server is in an inconsistent state, it can always accept a connection from a client, but risks never sending it a response over the socket. For avoiding these blocking waits, it is useful to fix a limit to the time for transmission of the response. After this time has elapsed, the client supposes that the server is no longer responding. Then the client can close this connection in order to go on to its other work. This is how WWW browsers work. When a request has no response after a certain time, the browser decides to indicate that to the user. Objective Caml has input-output with time limits. In the `Thread` library, the functions `wait_time_read` and `wait_time_write` suspend execution until a character can be read or written, within a certain time limit. As input, these function take a file descriptor and a time limit in seconds: *`Unix.file_descr -> float -> bool`*. If the time limit has passed, the function returns `false`, otherwise the I/O is processed.

Transmitting Values in their Internal Representation

The interest in transmission of internal values comes from simplifying the protocol. There is no longer any need to encode and decode data in a textual format. The inherent difficulty in sending and receiving values in their internal representation are the same as those encountered for persistent values (see the `Marshal` library, page 228). In effect, reading or writing a value in a file is equivalent to receiving the same value over a socket.

Functional Values

In the case of transmitting a closure between two Objective Caml programs, the code in the closure is not sent, only its environment and its code pointer (see figure 12.9 page 334). For this strategy to work, it is necessary that the server possess the same code in the same memory location. This implies that the same program is running on the server as on the client. Nothing, however, prevents the two programs from running different parts of the code at the same time. We adapt the matrix calculation service by sending a closure with an environment containing the data for calculation. When it is received, the server applies this closure to () and the calculation begins.

Interoperating with Different Languages

The interest in text protocols is that they are independent of implementation languages for clients and servers. In effect, the ASCII code is always known by programming languages. Therefore, it is up to the client and to the server to analyze syntactically the strings of characters transmitted. An example of such an open protocol is the simulation of soccer players called RoboCup.

Soccer Robots

A soccer team plays against another team. Each member of the team is a client of a referee server. The players on the same team cannot communicate directly with each other. They must send information through the server, which retransmits the dialog. The server shows a part of the field, according to the player's position. All these communications follow a text protocol. A Web page that describes the protocol, the server, and certain clients:

Link: http://www.robocup.org/

The server is written in C. The clients are written in different languages: C, C++, SmallTalk, Objective Caml, etc. Nothing prevents a team from fielding players written in different languages.

This protocol responds to the interoperability needs between programs in different implementation languages. It is relatively simple, but it requires a particular syntax analyzer for each family of languages.

Exercises

The suggested exercises allow you to try different types of distributed applications. The first offers a new network service for setting the time on client machines. The second exercise shows how to use resources on different machines to distribute a calculation.

Service: Clock

This exercise consists of implementing a "clock" service that gives the time to any client. The idea is to have a reference machine to set the time for different machines on a network.

1. Define a protocol for transmitting a date containing the day, month, hour, minute, and second.
2. Write the function or the class for the service reusing one of the generic servers presented in the Chapter. The service sends date information over each accepted connection, then closes the socket.
3. Write the client , which sets the clock every hour.
4. Keep track of time differences when requests are sent.

A Network Coffee Machine

We can build a little service that simulates a beverage vending machine. A summary description of the protocol between the client and service is as follows:

- when it makes a connection, the client receives a list of available drinks;
- it then sends to the server its beverage choice;
- the server returns the price of the beverage;
- the client sends the requested price, or some other sum;
- the server responds with the name of the chosen beverage and shows the change tendered.

The server may also respond with an error message if it has not understood a request, does not have enough change, etc. A client request always contains just one piece of information.

The exchanges between client and server are in the form of strings of characters. The different components of a message are separated by two periods and all strings end in `:$\n`.

The service function communicates with the coffee machine by using a file to pass commands and a hash table for recovering drinks and change.

This exercise will make use of sockets, lightweight processes with a little concurrency, and objects.

1. Rewrite the function `establish_server` using the primitives in `ThreadUnix`.
2. Write two functions, `get_request` and `send_answer`. The first function reads and encodes a request and the second formats and sends a response beginning with a list of strings of characters.
3. Write a class `cmd_fifo` to manage pending commands. Each new command is assigned a unique number. For this purpose, implement a class `num_cmd_gen`.

4. Write a class `ready_table` for stocking the machine with drinks.
5. Write the class `machine` that models the coffee machine. The class contains a method `run` that loops through the sequence: wait for a command, then execute it, as long as there remain drinks available. Define a type *`drink_descr`* indicating, for each drink: its name, the quantity in stock, the quantity that will remain after satisying pending commands, and its price. We can use an auxiliary function `array_index` which returns the index of the first element in a table satisfying a criterion passed as a parameter.
6. Write the service function `waiter`.
7. Write the principal function `main` that obtains a port number for the service from the command line and performs a number of initialization tasks. In particular, the coffee machine executes in a process.

Summary

This chapter presented the new possibilities offered by distributed programming. Communication between programs is accomplished with the fundamental mechanism of sockets, used by low-level Internet protocols. The action models used by clients and servers are asymmetric. Communication between clients and servers use some notion of protocol, most often using plain text. Functional programming and object-oriented programming allow us to easily build distributed applications. The client-server model lends itself to different software architectures, with two or three tiers, according to the distribution of tasks between them.

To Learn More

Communication between distant Objective Caml programs can be rich. Use of text protocols is greatly facilitated by utilities for syntactic analysis (see Chapter 11). The persistence mechanism offered by the `Marshal` library (see Chapter 8) allows sending complex data in its internal format including functional values if the two communicating programs are the same. The main deficiency of that mechanism is the inability to send instances of classes. One solution to that problem is to use an ORB (Object Request Broker) to transmit objects or invoke remote methods. This architecture already exists in many object-oriented languages in the form of the CORBA (Common ORB Architecture) standard. This standard from the OMG (Object Management Group), which debuted in 1990, allows the use of remote objects, and is independent of the implementation language used to create classes.

Link: http://www.omg.org

The two principal functions of CORBA are the ability to send objects to a remote program and, especially, the ability to use the same object at many locations in a network, in order to call methods which can modify its instance variables. Further, this standard is independent of the language used to implement these remote objects. To

that end, an ORB furnishes a description language for interfaces called IDL (Interface Definition Language), in the manner of `CAMLIDL` for the interface between Objective Caml and C. For the moment, there is no ORB that works with Objective Caml, but it is possible to build one, since the IDL language is an abstraction of object-oriented languages with classes. To simplify, CORBA furnishes a software bus (IIOP) that allows transferring and addressing remote data.

The ability to reference the same object at many points in a network simulates distributed shared memory, which is not without problems for automatic garbage collection.

The ability to reference a remote object does not cause code to be transferred. One can only receive a copy of an instance of a class if the class exists on the server. For certain client-server applications, it may be necessary to use dynamic loading of code (such as in Java applets) and even to migrate processes along with their code. An interesting example of dynamic loading of remote code is the MMM browser built in Objective Caml by François Rouaix:

Link: http://caml.inria.fr/~rouaix/mmm/

This browser can be used conventionally to view Web pages, but can also load Objective Caml applets from a server and run them in a graphical window.