21

Applications

The first application is really a toolbox to facilitate the construction of client-server applications which transmit Objective Caml values. To build an application using the toolbox, one need only implement serialization functions for the values to be transmitted, then apply a functor to obtain an abstract class for the server, then add the application's processing function by means of inheritance.

The second application revisits the robot simulation, presented on page 550, and adapts it to the client-server model. The server represents the world in which the robot clients move around. We thus simulate distributed memory shared by a group of clients possibly located on various machines on the network.

The third application is an implementation of some small HTTP servers (called *servlets*). A server knows how to respond to an HTTP request such as a request to retrieve an HTML page. Moreover, it is possible to pass values in these requests using the CGI format of HTTP servers. We will use this functionality right away to construct a server for requests on the association database, described on page 148. As a client, we will use a Web browser to which we will send an initial page containing the query form.

Client-server Toolbox

We present a collection of modules to enable client-server interactions among Objective Caml programs. This toolbox will be used in the two applications that follow.

A client-server application differs from others in the protocol that it uses and in the processing that it associates with the protocol. Otherwise, all such applications use very similar mechanisms: waiting for a connection, starting a separate process to handle the connection, and reading and writing sockets.

Taking advantage of Objective Caml's ability to combine modular genericity and extension of objects, we will create a collection of functors which take as argument a

communications protocol and produce generic classes implementing the mechanisms of clients and of servers. We can then subclass these to obtain the particular processing we need.

Protocols

A communications protocol is a type of data that can be translated into a sequence of characters and transmitted from one machine to another via a socket. This can be described using a signature.

```
# module type PROTOCOL =
    sig
      type t
      val to_string : t → string
      val of_string : string → t
    end ;;
```

The signature requires that the data type be monomorphic; yet we can choose a data type as complex as we wish, as long as we can translate it to a sequence of characters and back. In particular, nothing prevents us from using objects as our data.

```
# module Integer =
    struct
      class integer x =
        object
          val v = x
          method x = v
          method str = string_of_int v
        end
      type t = integer
      let to_string o = o#str
      let of_string s = new integer (int_of_string s)
    end ;;
```

By making some restrictions on the types of data to be manipulated, we can use the module `Marshal`, described on page 229, to define the translation functions.

```
# module Make_Protocol = functor ( T : sig type t end ) →
    struct
      type t = T.t
      let to_string (x:t) = Marshal.to_string x [Marshal.Closures]
      let of_string s = (Marshal.from_string s 0 : t)
    end ;;
```

Communication

Since a protocol is a type of value that can be translated into a sequence of characters, we can make these values persistent and store them in a file.

The only difficulty in reading such a value from a file when we do not know its type is that *a priori* we do not know the size of the data in question. And since the file in question is in fact a socket, we cannot simply check an end of file marker. To solve this problem, we will write the size of the data, as a number of characters, before the data itself. The first twelve characters will contain the size, padded with spaces.

The functor `Com` takes as its parameter a module with signature `PROTOCOL` and defines the functions for transmitting and receiving values encoded using the protocol.

```
# module Com = functor (P : PROTOCOL) →
    struct
      let send fd m =
        let mes = P.to_string m in
        let l = (string_of_int (String.length mes)) in
        let buffer = String.make 12 ' ' in
          for i=0 to (String.length l)-1 do buffer.[i] <- l.[i] done ;
          ignore (ThreadUnix.write fd buffer 0 12) ;
          ignore (ThreadUnix.write fd mes 0 (String.length mes))

      let receive fd =
        let buffer = String.make 12 ' '
        in
          ignore (ThreadUnix.read fd buffer 0 12) ;
          let l = let i = ref 0
          in while (buffer.[!i]<>' ') do incr i done ;
             int_of_string (String.sub buffer 0 !i)
        in
          let buffer = String.create l
          in ignore (ThreadUnix.read fd buffer 0 l) ;
             P.of_string buffer
    end ;;
module Com :
  functor(P : PROTOCOL) ->
    sig
      val send : Unix.file_descr -> P.t -> unit
      val receive : Unix.file_descr -> P.t
    end
```

Note that we use the functions `read` and `write` from module `ThreadUnix` and not those from module `Unix`; this will permit us to use our functions in a thread without blocking the execution of other processes.

Server

A server is built as an abstract class parameterized by the type of data in the protocol. Its constructor takes as arguments a port number and the maximum number of simultaneous connections allowed. The method for processing a request is abstract; it must be implemented in a subclass of `server` to obtain a concrete class.

```
# module Server = functor (P : PROTOCOL) →
    struct
      module Com = Com (P)

      class virtual ['a] server p np =
        object (s)
          constraint 'a = P.t
          val port_num = p
          val nb_pending = np
          val sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0

          method start =
            let host = Unix.gethostbyname (Unix.gethostname()) in
            let h_addr = host.Unix.h_addr_list.(0) in
            let sock_addr = Unix.ADDR_INET(h_addr, port_num) in
              Unix.bind sock sock_addr ;
              Unix.listen sock nb_pending ;
              while true do
                let (service_sock, client_sock_addr) = ThreadUnix.accept sock
                in ignore (Thread.create s#process service_sock)
              done
          method send = Com.send
          method receive = Com.receive
          method virtual process : Unix.file_descr → unit
        end
    end ;;
```

In order to show these ideas in use, let us revisit the CAPITAL service, adding the capability of sending lists of strings.

```
# type message = Str of string | LStr of string list ;;
# module Cap_Protocol = Make_Protocol (struct type t=message end) ;;
# module Cap_Server = Server (Cap_Protocol) ;;

# class cap_server p np =
    object (self)
      inherit [message] Cap_Server.server p np
      method process fd =
        match self#receive fd with
            Str s  → self#send fd (Str (String.uppercase s)) ;
                     Unix.close fd
          | LStr l → self#send fd (LStr (List.map String.uppercase l)) ;
                     Unix.close fd
    end ;;
class cap_server :
  int ->
  int ->
  object
    val nb_pending : int
    val port_num : int
    val sock : Unix.file_descr
    method process : Unix.file_descr -> unit
```

```
    method receive : Unix.file_descr -> Cap_Protocol.t
    method send : Unix.file_descr -> Cap_Protocol.t -> unit
    method start : unit
  end
```

The processing consists of receiving a request, examining it, processing it and sending the result. The functor allows us to concentrate on this processing while constructing the server; the rest is generic. However, if we wanted a different mechanism, such as for example using acknowledgements, nothing would prevent us from redefining the inherited methods for communication.

Client

To construct clients using a given protocol, we define three general-purpose functions:

- `connect`: establishes a connection with a server; it takes the address (IP address and port number) and returns a file descriptor corresponding to a socket connected to the server.
- `emit_simple`: opens a connection, sends a message and closes the connection.
- `emit_answer`: same as `emit_simple`, but waits for the server's response before closing the connection.

```
# module Client = functor (P : PROTOCOL) →
    struct
      module Com = Com (P)

      let connect addr port =
        let sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0
        and in_addr = (Unix.gethostbyname addr).Unix.h_addr_list.(0)
        in ThreadUnix.connect sock (Unix.ADDR_INET(in_addr, port)) ;
           sock

      let emit_simple addr port mes =
        let sock = connect addr port
        in Com.send sock mes ;  Unix.close sock

      let emit_answer addr port mes =
        let sock = connect addr port
        in Com.send sock mes ;
           let res = Com.receive sock
           in Unix.close sock ; res
    end ;;
module Client :
  functor(P : PROTOCOL) ->
    sig
      module Com :
        sig
          val send : Unix.file_descr -> P.t -> unit
```

```
        val receive : Unix.file_descr -> P.t
      end
    val connect : string -> int -> Unix.file_descr
    val emit_simple : string -> int -> P.t -> unit
    val emit_answer : string -> int -> P.t -> P.t
  end
```

The last two functions are of a higher level than the first: the mechanism linking the client and the server does not appear. The caller of `emit_answer` does not even need to know that the computation it is requesting is carried out by a remote machine. As far as the caller is concerned, it invokes a function that is represented by an address and port, with an argument which is the message to be sent, and a value is returned to it. The distributed aspect can seem entirely hypothetical.

A client of the CAPITAL service is extremely easy to construct. Assume that the `boulmich` machine provides the service on port number 12345; then the function `list_uppercase` can be defined by means of a call to the service.

```
# let list_uppercase l =
    let module Cap_client = Client (Cap_Protocol)
    in match Cap_client.emit_answer "boulmich" 12345 (LStr l)
       with Str x → [x]
         | LStr x → x ;;
val list_uppercase : string list -> string list = <fun>
```

To Learn More

The first improvement to be made to our toolbox is some error handling, which has been totally absent so far. Recovery from exceptions which arise from a broken connection, and a mechanism for retrying, would be most welcome.

In the same vein, the client and the server would benefit from a timeout mechanism which would make it possible to limit the time to wait for a response.

Because we have constructed the generic server as a class, which moreover is parameterized by the type of data to be transmitted over the network, it is easy to extend it to augment or modify its behavior in order to implement any desired improvements.

Another approach is to enrich the communication protocols. One can for example add requests for acknowledgement to the protocol, or accompany each request by a checksum allowing verification that the network has not corrupted the data.

The Robots of Dawn

As we promised in the last application of the third part (page 550), we will now revisit the problem of robots in order to treat it in a distributed framework where the world is a server and where each robot is an independent process capable of being executed on a remote machine.

This application is a good summary of the possibilities of the Objective Caml language because we will utilize and combine the majority of its features. In addition to the distributed model which is imposed on us by the exercise, we will make use of concurrency to construct a server in which multiple connections will be handled independently while all sharing a single memory representation of the "world". All access to and modification of the state of affairs of the world will therefore have to be protected by critical sections.

In order to reuse as much as possible the code that we have already built for robots in one section, and the client-server architecture of another section, we will use functors and inheritance of classes at the same time.

This application is quite minimal, but we will see that its architecture lends itself particularly well to extensions in multiple directions.

World-Server

We take a representation of the world similar to that which we developed in Part III. The world is a grid of finite size, and each cell of the grid can be occupied by only one robot. A robot is identified by its name and by its position; the world is determined by its size and by the robots that live in it. This information is represented by the following types:

```
# type position = { x:int ; y:int } ;;

# type robot_info = { name : string ; mutable pos : position }
  type world_info = { length : int ; width  : int ;
                      mutable robots : robot_info list } ;;
```

The world will have to serve two sorts of clients:

- passive clients which simply observe the positions of various robots. They will allow us to build the clients in charge of displays. We will call them *spies*.
- active clients, able to ask the server to move robots and thus modify its state.

These two categories of clients and their behavior will determine the collection of messages exchanged by the server and clients.

When a client connects, it declares itself passive (`Spy`) or active (`Enter`). A spy receives as response to its connection the global state of the world. Then, it is kept informed of all changes. However, it cannot submit any requests. A robot which connects must supply its characteristics (its name and its initial position); the world then confirms its arrival. Then, it can request information: its own position (`GetPos`) or the list of robots that surround it (`Look`). It can also instruct the world to move it. The protocol of requests to the world from distributed robots is represented by the following type:

```
# type query =
    | Spy                   (* initial declaration requests *)
    | Enter of robot_info
```

```
    | Move of position       (* robot requests *)
    | GetPos
    | Look of int

    | World of world_info    (* messages delivered by the world *)
    | Pos of robot_info
    | Exit of robot_info ;;
```

From this protocol, using the functors from the "distributed toolbox" of the previous chapter, we immediately derive the generic server.

```
# module Pquery = Make_Protocol (struct type t = query end ) ;;
# module Squery = Server (Pquery) ;;
```

Now we need only specify the behavior of the server by implementing the method `process` to handle both the data that represent the world and the data for managing connections.

More precisely, the server contains a variable `world` (of type *world_info*) which is protected by the lock `sem` (of type *Mutex.t*). It also contains a variable `spies` which is a list of queues of messages to send to observers, with one queue per spy. To activate the processes in charge of sending these messages, the server also maintains a signal (of type *Condition.t*).

We provide an auxiliary function `dist` to calculate the distance between two positions:

```
# let dist p q = max (abs (p.x-q.x)) (abs (p.y-q.y)) ;;
val dist : position -> position -> int = <fun>
```

The function `critical` encapsulates the calculation of a value within a critical section:

```
# let critical m f a =
    Mutex.lock m ; let r = f a in Mutex.unlock m ; r ;;
val critical : Mutex.t -> ('a -> 'b) -> 'a -> 'b = <fun>
```

Here is the definition of the class `server` implementing the world-server. It is long, but we will follow it up with a step-by-step explanation.

```
# class  server l w n np =
    object (self)
      inherit [query] Squery.server n np
      val world = { length=l ; width=w ; robots=[] }
      val sem = Mutex.create ()
      val mutable spies = []
      val signal = Condition.create ()

      method lock = Mutex.lock sem
      method unlock = Mutex.unlock sem
```

```
method legal_pos p = p.x>=0 && p.x<l && p.y>=0 && p.y<w

method free_pos p =
  let is_not_here r = r.pos.x<>p.x || r.pos.y<>p.y
  in critical sem (List.for_all is_not_here) world.robots

method legal_move r p =
  let dist1 p = (dist r.pos p) <= 1
  in (critical sem dist1 p) && self#legal_pos p && self#free_pos p


method queue_message mes =
  List.iter (Queue.add mes) spies ;
  Condition.broadcast signal

method trace_loop s q =
  let foo = Mutex.create () in
  let f () =
    try
      spies <- q :: spies ;
      self#send s (World world) ;
      while true do
        while Queue.length q = 0 do Condition.wait signal foo done ;
        self#send s (Queue.take q)
      done
    with _ → spies <- List.filter ((!=) q) spies ;
             Unix.close s
  in ignore (Thread.create f ())

method remove_robot r =
  self#lock ;
  world.robots <- List.filter ((<>) r) world.robots ;
  self#queue_message (Exit {r with name=r.name}) ;
  self#unlock

method try_move_robot r p =
  if self#legal_move r p
  then begin
         self#lock ;
         r.pos <- p ;
         self#queue_message (Pos {r with name=r.name}) ;
         self#unlock
       end

method process_robot s r =
  let f () =
    try
      world.robots <- r :: world.robots ;
      self#send s (Pos r) ;
      self#queue_message (Pos r) ;
      while true do
```

```
              Thread.delay 0.5 ;
              match self#receive s with
                  Move p → self#try_move_robot r p
                | GetPos → self#send s (Pos r)
                | Look d →
                    self#lock ;
                    let dist p = max (abs (p.x-r.pos.x)) (abs (p.y-r.pos.y)) in
                    let l = List.filter (fun x → (dist x.pos)<=d) world.robots
                    in self#send s (World { world with robots = l }) ;
                       self#unlock
                | _ → ()
            done
          with  _ → self#unlock ;
                    self#remove_robot r ;
                    Unix.close s
        in ignore (Thread.create f ())

      method process s =
        match self#receive s with
            Spy → self#trace_loop s (Queue.create ())
          | Enter r →
            ( if not (self#legal_pos r.pos && self#free_pos r.pos) then
              let i = ref 0 and j = ref 0 in
              ( try
                  for x=0 to l do
                    for y=0 to w do
                      let p = { x=x ; y=y }
                      in if self#legal_pos p && self#free_pos p
                         then ( i:=x ; j:=y; failwith "process" )
                    done done ;
                  Unix.close s
                with Failure "process" → r.pos <- { x= !i ; y= !j }  )) ;
              self#process_robot s r
          | _ → Unix.close s

    end ;;
class server :
  int ->
  int ->
  int ->
  int ->
  object
    val nb_pending : int
    val port_num : int
    val sem : Mutex.t
    val signal : Condition.t
    val sock : Unix.file_descr
    val mutable spies : Pquery.t Queue.t list
    val world : world_info
    method free_pos : position -> bool
    method legal_move : robot_info -> position -> bool
    method legal_pos : position -> bool
```

```
    method lock : unit
    method process : Unix.file_descr -> unit
    method process_robot : Unix.file_descr -> robot_info -> unit
    method queue_message : Pquery.t -> unit
    method receive : Unix.file_descr -> Pquery.t
    method remove_robot : robot_info -> unit
    method send : Unix.file_descr -> Pquery.t -> unit
    method start : unit
    method trace_loop : Unix.file_descr -> Pquery.t Queue.t -> unit
    method try_move_robot : robot_info -> position -> unit
    method unlock : unit
  end
```

The method `process` starts out by distinguishing between the two types of client. Depending on whether the client is active or passive, it invokes a processing method called: `trace_loop` for an observer, `process_robot` for a robot. In the second case, it checks that the initial position proposed by the client is compatible with the state of the world; if not, it finds a valid initial position. The remainder of the code can be divided into four categories:

1. **General methods**: these are methods which we developed in Part III for general worlds. Mainly, it is a matter of verifying that a displacement is legal for a given robot.
2. **Management of observers**: each observer is associated with a socket through which it is sent data, with a queue containing all the messages which have not yet been sent to it, and with a process. The method `trace_loop` is an infinite loop that empties the queue of messages by sending them; it goes to sleep when the queue is empty. The queues are filled, all at the same time, by the method `queue_message`. Note that after appending a message, the activation signal is sent to all processes.
3. **Management of robots**: here again, each robot is associated with a dedicated process. The method `process_robot` is an infinite loop: it waits for a request, processes it, and responds if necessary. Then it resumes waiting for the next request. Note that it is these robot-management methods which issue calls to the method `queue_message` when the state of the world has been modified. If the connection with a robot is lost—that is, if an exception is raised while waiting for a request—the robot is considered to have terminated and its departure is signaled to the observers.
4. **Inherited methods**: these are the methods of the generic server obtained by application of the functor `Server` to the protocol of our application.

Observer-client

The functor `Client` gives us generic functions for connecting with a server according to the particular protocol that concerns us here.

```
# module Cquery = Client (Pquery) ;;
module Cquery :
```

```
  sig
    module Com :
      sig
        val send : Unix.file_descr -> Pquery.t -> unit
        val receive : Unix.file_descr -> Pquery.t
      end
    val connect : string -> int -> Unix.file_descr
    val emit_simple : string -> int -> Pquery.t -> unit
    val emit_answer : string -> int -> Pquery.t -> Pquery.t
  end
```

The behavior of a spy is simple: it connects to the server and displays the information that the server sends it. The spy includes three display functions which we provide below:

```
# let display_robot r =
    Printf.printf "The robot %s is located at (%d,%d)\n" r.name r.pos.x r.pos.y ;
    flush stdout ;;
val display_robot : robot_info -> unit = <fun>

# let display_exit r =   Printf.printf "The robot %s has terminated\n" r.name ;
    flush stdout ;;
val display_exit : robot_info -> unit = <fun>

# let display_world w =
    Printf.printf "The world is a grid of size %d by %d \n" w.length w.width ;
    List.iter display_robot w.robots ;
    flush stdout ;;
val display_world : world_info -> unit = <fun>
```

The primary function of the spy-client is:

```
# let trace_client name port =
    let sock = Cquery.connect name port
    in Cquery.Com.send sock Spy ;
       ( match Cquery.Com.receive sock with
            World w → display_world w
          | _ → failwith "the server did not follow the protocol" ) ;
       while true do
         match Cquery.Com.receive sock with
            Pos r → display_robot r
          | Exit r → display_exit r
          |_ → failwith "the server did not follow the protocol"
       done ;;
val trace_client : string -> int -> unit = <fun>
```

There are two ways of constructing a graphical display. The first is simple but not very efficient: since the server sends the complete set of information when a connection is established, one can simply open a new connection at regular intervals, display the world in its entirety, and close the connection. The other approach involves using the information sent by the server to maintain a copy of the state of the world. It is then

easy to display only the modifications to the state upon reception of messages. It is this second solution which we have implemented.

Robot-Client

As we defined them in the previous chapter (cf. page 550), the robots conform to the following signature.

```
# module type ROBOT =
    sig
      class robot : int → int →
        object
          val mutable i : int
          val mutable j : int
          method get_pos : int * int
          method next_pos : unit → int * int
          method set_pos : int * int → unit
        end
      end ;;
```

The part that we wish to save from the various classes is that which necessarily varies from one type of robot to another and which defines its behavior: the method `next_pos`.

In addition, we need a method for connecting the robot to the world (`start`) and a loop that alternately calculates a new position and communicates with the server to submit the chosen position.

We define a functor which, when given a class implementing a virtual robot (that is, conforming to the signature `ROBOT`), creates, by inheritance, a new class containing the proper methods to make an autonomous client out of the robot.

```
# module RobotClient (R : ROBOT) =
    struct
      class robot robname x y hostname port =
        object (self)
          inherit R.robot x y as super
          val mutable socket = Unix.stderr
          val mutable rob = { name=robname ; pos={x=x;y=y} }

          method private adjust_pos r =
            rob.pos <- r.pos ; i <- r.pos.x ; j <- r.pos.y

          method get_pos =
            Cquery.Com.send socket GetPos ;
            match Cquery.Com.receive socket with
                Pos r → self#adjust_pos r ; super#get_pos
              | _ → failwith "the server did not follow the protocol"
```

```
          method set_pos =
            failwith "the method set_pos cannot be used"

          method start =
            socket <- Cquery.connect hostname port ;
            Cquery.Com.send socket (Enter rob) ;
            match Cquery.Com.receive socket with
                Pos r → self#adjust_pos r ; self#run
              | _ → failwith "the server did not follow the protocol"

          method run =
            while true do
              let (x,y) = self#next_pos ()
              in Cquery.Com.send socket (Move {x=x;y=y}) ;
                 ignore (self#get_pos)
            done
        end
    end ;;
module RobotClient :
  functor(R : ROBOT) ->
    sig
      class robot :
        string ->
        int ->
        int ->
        string ->
        int ->
        object
          val mutable i : int
          val mutable j : int
          val mutable rob : robot_info
          val mutable socket : Unix.file_descr
          method private adjust_pos : robot_info -> unit
          method get_pos : int * int
          method next_pos : unit -> int * int
          method run : unit
          method set_pos : int * int -> unit
          method start : unit
        end
    end
```

Notice that the method `get_pos` has been redefined as a query to the server: the instance variables `i` and `j` are not reliable, because they can be modified without the consent of the world. For the same reason, the use of `set_pos` has been made invalid: calling it will always raise an exception. This policy may seem severe, but it's a good bet that if this method were used by `next_pos` then a discrepancy would appear between the real position (as known by the server) and the supposed position (as known by the client).

We use the functor `RobotClient` to create various classes corresponding to the various robots.

```
# module Fix = RobotClient (struct class robot = fix_robot end) ;;
# module Crazy = RobotClient (struct class robot = crazy_robot end) ;;
# module Obstinate = RobotClient (struct class robot = obstinate_robot end) ;;
```

The following small program provides a way to launch the server and the various clients from the command line. The argument passed to the program specifies which one to launch.

```
# let port = 1200 in
    if Array.length Sys.argv >=2 then
      match Sys.argv.(1) with
        "1" → let s = new server 25 30 port 10 in s#start
      | "2" → trace_client "localhost" port
      | "3" → let o = new Fix.robot "fix" 10 10 "localhost" port in o#start
      | "4" → let o = new Crazy.robot "crazy" 10 10 "localhost" port in o#start
      | "5" → let o = new Obstinate.robot "obstinate" 10 10 "localhost" port
              in o#start
      | _ → () ;;
```

To Learn More

The world of robots stimulates the imagination. With the elements already given here, one can easily create an "intelligent robot" which is both a robot and a spy. This allows the various inhabitants of the world to cooperate. One can then extend the application to obtain a small action game like "chickens-foxes-snakes" in which the foxes chase the chickens, the snakes chase the foxes and the chickens eat the snakes.

HTTP Servlets

A *servlet* is a "module" that can be integrated into a server application to respond to client requests. Although a servlet need not use a specific protocol, we will use the HTTP protocol for communication (see figure 21.1). In practice, the term servlet refers to an HTTP servlet.

The classic method of constructing dynamic HTML pages on a server is to use CGI (Common Gateway Interface) commands. These take as argument a URL which can contain data coming from an HTML form. The execution then produces a new HTML page which is sent to the client. The following links describe the HTTP and CGI protocols.

Link: http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1945.html

Link: http://hoohoo.ncsa.uiuc.edu/docs/cgi/overview.html

It is a slightly heavyweight mechanism because it launches a new program for each request.

HTTP servlets are launched just once, and can can decode arguments in CGI format to execute a request. Servlets can take advantage of the Web browser's capabilities to construct a graphical interface for an application.

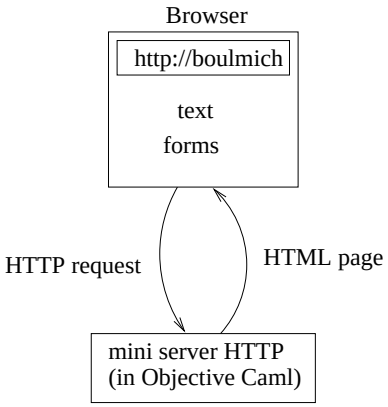

Figure 21.1: communication between a browser and an Objective Camlserver

In this section we will define a server for the HTTP protocol. We will not handle the entire specification of the protocol, but instead will limit ourselves to those functions necessary for the implementation of a server that mimics the behavior of a CGI application.

At an earlier time, we defined a generic server module `Gsd`. Now we will give the code to create an application of this generic server for processing part of the HTTP protocol.

HTTP and CGI Formats

We want to obtain a server that imitates the behavior of a CGI application. One of the first tasks is to decode the format of HTTP requests with CGI extensions for argument passing.

The clients of this server can be browsers such as Netscape or Internet Explorer.

Receiving Requests

Requests in the HTTP protocol have essentially three components: a method, a URL and some data. The data must follow a particular format.

In this section we will construct a collection of functions for reading, decomposing and decoding the components of a request. These functions can raise the exception:

```
# exception Http_error of string ;;
exception Http_error of string
```

Decoding The function `decode`, which uses the helper function `rep_xcode`, attempts to restore the characters which have been encoded by the HTTP client: spaces (which have been replaced by `+`), and certain reserved characters which have been replaced by their hexadecimal code.

```
# let rec rep_xcode s i =
    let xs = "0x00" in
      String.blit s (i+1) xs 2 2;
      String.set s i (char_of_int (int_of_string xs));
      String.blit s (i+3) s (i+1) ((String.length s)-(i+3));
      String.set s ((String.length s)-2) '\000';
      Printf.printf"rep_xcode1(%s)\n" s ;;
val rep_xcode : string -> int -> unit = <fun>

# exception End_of_decode of string ;;
exception End_of_decode of string

# let decode s =
    try
      for i=0 to pred(String.length s) do
        match s.[i] with
          '+' → s.[i] <- ' '
        | '%' → rep_xcode s i
        | '\000' → raise (End_of_decode (String.sub s 0 i))
        | _ → ()
      done;
      s
    with
     End_of_decode s → s ;;
val decode : string -> string = <fun>
```

String manipulation functions The module `String_plus` contains some functions for taking apart character strings:

- `prefix` and `suffix`, which extract the substrings to either side of an index;
- `split`, which returns the list of substrings determined by a separator character;
- `unsplit`, which concatenates a list of strings, inserting separator characters between them.

```
# module String_plus =
   struct
    let prefix s n =
     try String.sub s 0 n
     with Invalid_argument("String.sub") → s
```

```
    let suffix s i =
     try String.sub s i ((String.length s)-i)
     with Invalid_argument("String.sub") → ""

    let rec split c s =
     try
      let i = String.index s c in
      let s1, s2 = prefix s i, suffix s (i+1) in
       s1 :: (split c s2)
     with
      Not_found → [s]

    let unsplit c ss =
     let f s1 s2 = match s2 with "" → s1 | _ → s1^(Char.escaped c)^s2 in
      List.fold_right f ss ""
   end ;;
```

Decomposing data from a form Requests typically arise from an HTML page containing a form. The contents of the form are transmitted as a character string containing the names and values associated with the fields of the form. The function `get_field_pair` transforms such a string into an association list.

```
# let get_field_pair s =
    match String_plus.split '=' s with
      [n;v] → n,v
    | _ → raise (Http_error ("Bad field format : "^s)) ;;
val get_field_pair : string -> string * string = <fun>

# let get_form_content s =
    let ss = String_plus.split '&' s in
      List.map get_field_pair ss ;;
val get_form_content : string -> (string * string) list = <fun>
```

Reading and decomposing The function `get_query` extracts the method and the URL from a request and stores them in an array of character strings. One can thus use a standard CGI application which retrieves its arguments from the array of command-line arguments. The function `get_query` uses the auxiliary function `get`. We arbitrarily limit requests to a maximum size of 2555 characters.

```
# let get =
    let buff_size = 2555 in
      let buff = String.create buff_size in
        (fun ic → String.sub buff 0 (input ic buff 0 buff_size)) ;;
val get : in_channel -> string = <fun>

# let query_string http_frame =
    try
      let i0 = String.index http_frame ' ' in
```

```
      let q0 = String_plus.prefix http_frame i0 in
        match q0 with
          "GET"
          → begin
               let i1 = succ i0 in
               let i2 = String.index_from http_frame i1 ' ' in
               let q = String.sub http_frame i1 (i2-i1) in
               try
                 let i = String.index q '?' in
                 let q1 = String_plus.prefix q i in
                 let q = String_plus.suffix q (succ i) in
                   Array.of_list (q0 :: q1 :: (String_plus.split ' ' (decode q)))
               with
                 Not_found → [|q0;q|]
             end
        | _ → raise (Http_error ("Unsupported method: "^q0))
    with e → raise (Http_error ("Unknown request: "^http_frame)) ;;
val query_string : string -> string array = <fun>

# let get_query_string ic =
   let http_frame = get ic in
     query_string http_frame;;
val get_query_string : in_channel -> string array = <fun>
```

The Server

To obtain a CGI pseudo-server, able to process only the `GET` method, we write the class `http_servlet`, whose argument `fun_serv` is a function for processing HTTP requests such as might have been written for a CGI application.

```
# module Text_Server = Server (struct type t = string
                                      let to_string x = x
                                      let of_string x = x
                                end);;

# module P_Text_Server (P : PROTOCOL) =
  struct
    module Internal_Server = Server (P)

    class http_servlet  n np fun_serv =
      object(self)
        inherit [P.t] Internal_Server.server n np

        method receive_h fd =
          let ic = Unix.in_channel_of_descr fd in
            input_line ic

        method process fd =
          let oc = Unix.out_channel_of_descr fd in (
            try
              let request = self#receive_h fd in
```

```
              let args = query_string request in
                fun_serv oc args;
            with
              Http_error s → Printf.fprintf oc "HTTP error : %s <BR>" s
            | _ → Printf.fprintf oc "Unknown error <BR>" );
            flush oc;
            Unix.shutdown fd Unix.SHUTDOWN_ALL
      end
  end;;
```

As we do not expect the servlet to communicate using Objective Caml's special internal values, we choose the type *string* as the protocol type. The functions `of_string` and `to_string` do nothing.

```
# module Simple_http_server =
    P_Text_Server (struct type t = string
                          let of_string x = x
                          let to_string x = x
                   end);;
```

Finally, we write the primary function to launch the service and construct an instance of the class *http_servlet*.

```
# let cgi_like_server port_num fun_serv =
   let sv =   new Simple_http_server.http_servlet port_num 3  fun_serv
   in sv#start;;
val cgi_like_server : int -> (out_channel -> string array -> unit) -> unit =
  <fun>
```

Testing the Servlet

It is always useful during development to be able to test the parts that are already built. For this purpose, we build a small HTTP server which sends the file specified in the HTTP request as is. The function `simple_serv` sends the file whose name follows the GET request (the second element of the argument array). The function also displays all of the arguments passed in the request.

```
# let send_file oc f =
      let ic = open_in_bin f in
      try
        while true do
          output_byte oc (input_byte ic)
        done
      with End_of_file → close_in ic;;
val send_file : out_channel -> string -> unit = <fun>

# let simple_serv oc args =
    try
      Array.iter (fun x → print_string (x^" ")) args;
      print_newline();
      send_file oc args.(1)
    with _ → Printf.printf "error\n";;
val simple_serv : out_channel -> string array -> unit = <fun>
```

```
# let run n = cgi_like_server n simple_serv;;
val run : int -> unit = <fun>
```

The command *run* `4003` launches this servlet on port 4003. In addition, we launch a browser to issue a request to load the page `baro.html` on port 4003. The figure 21.2 shows the display of the contents of this page in the browser.

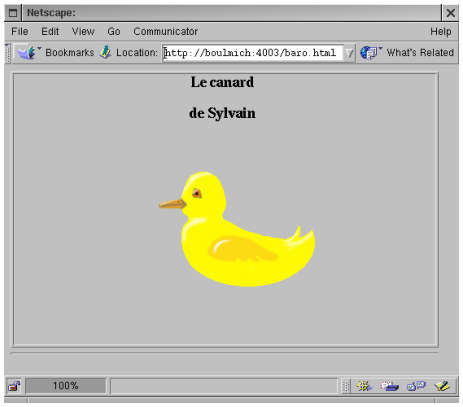

Figure 21.2: HTTP request to an Objective Caml servlet

The browser has sent the request `GET /baro.html` to load the page, and then the request `GET /canard.gif` to load the image.

HTML Servlet Interface

We will use a CGI-style server to build an HTML-based interface to the database of chapter 6 (see page 148).

The menu of the function `main` will now be displayed in a form on an HTML page, providing the same selections. The responses to requests are also HTML pages, generated dynamically by the servlet. The dynamic page construction makes use of the utilities defined below.

Application Protocol

Our application will use several elements from several protocols:

1. Requests are transmitted from a Web browser to our application server in the HTTP request format.
2. The data items within a request are encoded in the format used by CGI applications.
3. The response to the request is presented as an HTML page.

4. Finally, the nature of the request is specified in a format specific to the application.

We wish to respond to three kinds of request: queries for the list of mail addresses, queries for the list of email addresses, and queries for the state of received fees between two given dates. We give these query types respectively the names:
`mail_addr`, `email_addr` and `fees_state`. In the last case, we will also transmit two character strings containing the desired dates. These two dates correspond to the values of the fields `start` and `end` on an HTML form.

When a client first connects, the following page is sent. The names of the requests are encoded within it in the form of HTML anchors.

```
<HTML>
<TITLE> association </TITLE>
<BODY>
<HR>
<H1 ALIGN=CENTER>Association</H1>
<P>
<HR>
<UL>
<LI>List of
<A HREF="http://freres-gras.ufr-info-p6.jussieu.fr:12345/mail_addr">
mail addresses
</A>
<LI>List of
<A HREF="http://freres-gras.ufr-info-p6.jussieu.fr:12345/email_addr">
email addresses
</A>
<LI>State of received fees<BR>
<FORM
 method="GET"
 action="http://freres-gras.ufr-info-p6.jussieu.fr:12345/fees_state">
Start date : <INPUT type="text" name="start" value="">
End date : <INPUT type="text" name="end" value="">
<INPUT name="action" type="submit" value="Send">
</FORM>
</UL>
<HR>
</BODY>
</HTML>
```

We assume that this page is contained in the file `assoc.html`.

HTML Primitives

The HTML utility functions are grouped together into a single class called `print`. It has a field specifying the output channel. Thus, it can be used just as well in a CGI

application (where the output channel is the standard output) as in an application using the HTTP server defined in the previous section (where the output channel is a network socket).

The proposed methods essentially allow us to encapsulate text within HTML tags. This text is either passed directly as an argument to the method in the form of a character string, or produced by a function. For example, the principal method `page` takes as its first argument a string corresponding to the header of the page[1], and as its second argument a function that prints out the contents of the page. The method `page` produces the tags corresponding to the HTML protocol.

The names of the methods match the names of the corresponding HTML tags, with additional options added in some cases.

```
# class print (oc0:out_channel) =
    object(self)
      val oc = oc0
      method flush () = flush oc
      method str =
        Printf.fprintf oc "%s"
      method page header (body:unit → unit) =
        Printf.fprintf oc "<HTML><HEAD><TITLE>%s</TITLE></HEAD>\n<BODY>" header;
        body();
        Printf.fprintf oc "</BODY>\n</HTML>\n"
      method p () =
        Printf.fprintf oc "\n<P>\n"
      method br () =
        Printf.fprintf oc "<BR>\n"
      method hr () =
        Printf.fprintf oc "<HR>\n"
      method hr () =
        Printf.fprintf oc "\n<HR>\n"
      method h i s =
        Printf.fprintf oc "<H%d>%s</H%d>" i s i
      method h_center i s =
        Printf.fprintf oc "<H%d ALIGN=\"CENTER\">%s</H%d>" i s i
      method form url (form_content:unit → unit) =
        Printf.fprintf oc "<FORM method=\"post\" action=\"%s\">\n" url;
        form_content ();
        Printf.fprintf oc "</FORM>"
      method input_text =
        Printf.fprintf oc
          "<INPUT type=\"text\" name=\"%s\" size=\"%d\" value=\"%s\">\n"
      method input_hidden_text =
        Printf.fprintf oc "<INPUT type=\"hidden\" name=\"%s\" value=\"%s\">\n"
      method input_submit =
        Printf.fprintf oc "<INPUT name=\"%s\" type=\"submit\" value=\"%s\">"
      method input_radio =
        Printf.fprintf oc "<INPUT type=\"radio\" name=\"%s\" value=\"%s\">\n"
      method input_radio_checked =
```

1. This header is generally displayed in the title bar of the browser window.

```
      Printf.fprintf oc
        "<INPUT type=\"radio\" name=\"%s\" value=\"%s\" CHECKED>\n"
    method option =
      Printf.fprintf oc "<OPTION> %s\n"
    method option_selected opt =
      Printf.fprintf oc "<OPTION SELECTED> %s" opt
    method select name options selected =
      Printf.fprintf oc "<SELECT name=\"%s\">\n" name;
      List.iter
        (fun s → if s=selected then self#option_selected s else self#option s)
        options;
      Printf.fprintf oc "</SELECT>\n"
    method options selected =
      List.iter
        (fun s → if s=selected then self#option_selected s else self#option s)
  end ;;
```

We will assume that these utilities are provided by the module `Html_frame`.

Dynamic Pages for Managing the Association Database

For each of the three kinds of request, the application must construct a page in response. For this purpose we use the utility module `Html_frame` given above. This means that the pages are not really constructed, but that their various components are emitted sequentially on the output channel.
We provide an additional (virtual) page to be returned in response to a request that is invalid or not understood.

Error page The function `print_error` takes as arguments a function for emitting an HTML page (*i.e.*, an instance of the class `print`) and a character string containing the error message.

```
# let print_error (print:Html_frame.print) s =
    let print_body() =
     print#str s; print#br()
    in
     print#page "Error" print_body ;;
val print_error : Html_frame.print -> string -> unit = <fun>
```

All of our functions for emitting responses to requests will take as their first argument a function for emitting an HTML page.

List of mail addresses To obtain the page giving the response to a query for the list of mail addresses, we will format the list of character strings obtained by the function `mail_addresses`, which was defined as part of the database (see page 157). We will

assume that this function, and all others directly involving requests to the database, have been defined in a module named `Assoc`.

To emit this list, we use a function for outputting simple lines:

```
# let print_lines (print:Html_frame.print) ls =
    let print_line l =  print#str l; print#br() in
     List.iter print_line ls ;;
val print_lines : Html_frame.print -> string list -> unit = <fun>
```

The function for responding to a query for the list of mail addresses is:

```
# let print_mail_addresses print db =
     print#page "Mail addresses"
                (fun () → print_lines print (Assoc.mail_addresses db))
   ;;
val print_mail_addresses : Html_frame.print -> Assoc.data_base -> unit =
  <fun>
```

In addition to the parameter for emitting a page, the function `print_mail_addresses` takes the database as its second parameter.

List of email addresses This function is built on the same principles as that giving the list of mail addresses, except that it calls the function `email_addresses` from the module `Assoc`:

```
# let print_email_addresses print db =
     print#page "Email addresses"
                (fun () → print_lines print (Assoc.email_addresses db)) ;;
val print_email_addresses : Html_frame.print -> Assoc.data_base -> unit =
  <fun>
```

State of received fees The same principle also governs the definition of this function: retrieving the data corresponding to the request (which here is a pair), then emitting the corresponding character strings.

```
# let print_fees_state print db d1 d2 =
    let ls, t = Assoc.fees_state db d1 d2 in
    let page_body() =
     print_lines print ls;
     print#str ("Total : "^(string_of_float t));
     print#br()
    in
     print#page "State of received fees" page_body ;;
val print_fees_state :
  Html_frame.print -> Assoc.data_base -> string -> string -> unit = <fun>
```

Analysis of Requests and Response

We define two functions for producing responses based on an HTTP request. The first (`print_get_answer`) responds to a request presumed to be formulated using the GET method of the HTTP protocol. The second alters the production of the answer according to the actual method that the request used.

These two functions take as their second argument an array of character strings containing the elements of the HTTP request as analyzed by the function `get_query_string` (see page 668). The first element of the array contains the method, the second the name of the database request.
In the case of a query for the state of received fees, the start and end dates for the request are contained in the two fields of the form associated with the query. The data from the form are contained in the third field of the array, which must be decomposed by the function `get_form_content` (see page 668).

```
# let print_get_answer print q db =
    match q.(1) with
    | "/mail_addr" → print_mail_addresses print db
    | "/email_addr" → print_email_addresses print db
    | "/fees_state"
      → let nvs = get_form_content q.(2) in
        let d1 = List.assoc "start" nvs
        and d2 = List.assoc "end" nvs in
          print_fees_state print db d1 d2
    | _ → print_error print ("Unknown request: "^q.(1)) ;;
val print_get_answer :
  Html_frame.print -> string array -> Assoc.data_base -> unit = <fun>

# let print_answer print q db =
    try
      match q.(0) with
       "GET" → print_get_answer print q db
      | _ → print_error print ("Unsupported method: "^q.(0))
    with
      e
      → let s = Array.fold_right (^) q "" in
        print_error print ("Something wrong with request: "^s) ;;
val print_answer :
  Html_frame.print -> string array -> Assoc.data_base -> unit = <fun>
```

Main Entry Point and Application

The application is a standalone executable that takes the port number as a parameter. It reads in the database before launching the server. The main function is obtained from the function `print_answer` defined above and from the generic HTTP server function `cgi_like_server` defined in the previous section (see page 670). The latter function is located in the module `Servlet`.

```
# let get_port_num() =
    if (Array.length Sys.argv) < 2 then 12345
    else
      try int_of_string Sys.argv.(1)
      with _ → 12345 ;;
val get_port_num : unit -> int = <fun>

# let main() =
    let db = Assoc.read_base "assoc.dat" in
    let assoc_answer oc q = print_answer (new Html_frame.print oc) q db in
      Servlet.cgi_like_server (get_port_num()) assoc_answer ;;
val main : unit -> unit = <fun>
```

To obtain a complete application, we combine the definitions of the display functions into a file `httpassoc.ml`. The file ends with a call to the function `main`:

```
main() ;;
```

We can then produce an executable named `assocd` using the compilation command:

```
ocamlc -thread -custom -o assocd unix.cma threads.cma \
       gsd.cmo servlet.cmo html_frame.cmo string_plus.cmo assoc.cmo \
       httpassoc.ml -cclib -lunix -cclib -lthreads
```

All that's left is to launch the server, load the HTML page[2] contained in the file `assoc.html` given at the beginning of this section (page 672), and click.

The figure 21.3 shows an example of the application in use. The browser establishes an initial connection with the servlet, which sends it the menu page. Once the entry fields are filled in, the user sends a new request which contains the data entered. The server decodes the request and calls on the association database to retrieve the desired information. The result is translated into HTML and sent to the client, which then displays this new page.

To Learn More

This application has numerous possible enhancements. First of all, the HTTP protocol used here is overly simple compared to the new versions, which add a header supplying the type and length of the page being sent. Likewise, the method `POST`, which allows modification of the server, is not supported.[3]

To be able to describe the type of a page to be returned, the servlet would have to support the MIME convention, which is used for describing documents such as those attached to email messages.

2. ... taking care to update the URL according to your machine
3. Nothing prevents one from using `GET` for this, but that does not correspond to the standard.

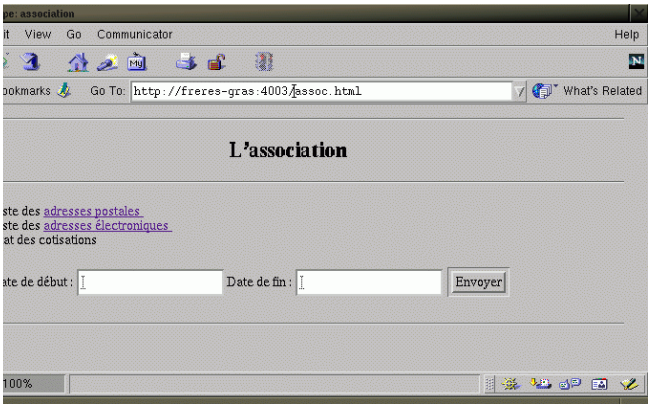

Figure 21.3: HTTP request to an Objective Caml servlet

The transmission of images, such as in figure 21.2, makes it possible to construct interfaces for 2-player games (see chapter 17), where one associates links with drawings of positions to be played. Since the server knows which moves are legal, only the valid positions are associated with links.

The MIME extension also allows defining new types of data. One can thus support a private protocol for Objective Caml values by defining a new MIME type. These values will be understandable only by an Objective Caml program using the same private protocol. In this way, a request by a client for a remote Objective Caml value can be issued via HTTP. One can even pass a serialized closure as an argument within an HTTP request. This, once reconstructed on the server side, can be executed to provide the desired result.