Jacques Garrigue, 2006 年 1 月 17 日

VI　ネットワークプログラミング

1　TCP/IP とソケット

まず，ネットワークを介したプログラム間通信の基本的な方法を見る．
ほとんどの OS が提供する二つの概念が中心になる．

1.1　IP アドレスとポート番号

まず，通信を行おうと思ったら，相手を指定する方法が提供されなければならない．人間の世界では住所 (または電話番号) と氏名が使われる．氏名だけだと，同姓同名があるかも知れないし，その人にどうやって連絡を取ればいいか分からない．住所だけだと，そこにいるどの人が受け取るべきか分からない．(携帯の電話番号は例外かも知れない)

IP アドレス　インターネットなどのネットワークでは，住所としては IP アドレスを使う．原則として，ネットワークにつながっている全てのコンピュータが異る IP アドレスを持っている．現在，主に使われている IPv4 では，IP アドレスは 32bit に整数で表現される．(moose.math.nagoya-u.ac.jp などの “.” で区切られた文字列は IP アドレスではない．ホスト名と言い，別の仕組みで IP アドレスに変換される．)

IP アドレスを書くとき，その 32bit の整数をそのまま書くのではなく，8bit ごとに値を区切る．例えば，前の moose というマシンの IP アドレスは “172.16.254.3” である．書くマシンの IP アドレスを調べるには，Unix/Linux では host か nslookup を使う．

```
$ host moose.math.nagoya-u.ac.jp
moose.math.nagoya-u.ac.jp has address 172.16.254.3
```

ちなみに，外のネットワークから上記のコマンドを使うと IP アドレスは 133.6.130.3 になる．同じマシンがいくつかの IP アドレスを持つことがある．少なくとも，全てのマシンは自分にしか分からないもう一つの IP アドレスを持っている．ループバック・アドレスと言われている 127.0.0.1 という特別の IP アドレスである．自分を指すアドレスなので，同じマシン内にあるものを指すことしかできない．

OCaml ではインターネットのアドレスは `Unix` モジュールの以下の関数によって扱われる．

```
type inet_addr                                  (* インターネットアドレスの抽象型 *)
val inet_addr_of_string : string -> inet_addr
                                        (* 整数 4 個からなるアドレスからの変換 *)
val string_of_inet_addr : inet_addr -> string                  (* 逆方向の変換 *)
val inet_addr_any : inet_addr                          (* bind 用の特別なアドレス *)
val inet_addr_loopback : inet_addr                                (* 127.0.0.1 *)
```

ポート番号　物理的な通信はコンピュータ間で行われるものだが，実際に通信しているのはプログラム同士 (あるいは，そのプログラムを利用している人間) である．IP アドレスだけではプログラムが特定できない．プログラムはどう指定すればいい．プログラムが実行されるときには，プロセス番号という一意的なものは割り振られるが，それは実行される度に変わるものなので使え

ない．そのプログラムのファイル名も考えられるが，プログラムは人間より合理的なので，通信するときに知りたいのは相手の名前ではなく，役割である．コンピュータによって，同じ役割を果すプログラムは違う名前になっているかも知れない．あるいは，同じプログラムは複数の役割を担っているかも知れない．そのために，あらかじめ決められてポート番号を使う．逆に，自分に役割があるのではなく，相手から呼び返して欲しい場合は，適当にまだこのマシンの誰もが使っていないポート番号を使えばいい．

役割を表しているポート番号をいくつか挙げる．Unix では，安全性のたみに 1023 以下のポート番号はユーザーが直接に束縛できないので，以下のポートで呼ばれるプログラムは管理者が決める．

ポート番号	サービス名	役割
21	ftp	データ転送の制御
22	ssh	セキュアシェルによる遠隔ログイン
23	telnet	遠隔ログイン
25	smtp	メールの送信先
53	domain	ホスト名変換サーバ
80	http	ウェブサーバ

1.2 TCP とソケット

通信プロトコルは大きく二つに分けられる．電話のように，通信が継続的にできるものと，手紙のように単発的に行われるもの．後者でも継続性のある動きを構築することができるが，多くの場合では前者のように，最初から継続性が与えられた方が便利である．Internet では前者の代表者は TCP と言い，双方向の継続した通信を提供している．しかも，データの全てが正しく配達されることを保証している．後者は UDP という，メッセージ単位で行われる通信方法を使う．UDP は正しく配達されることを保証しないが，仕事が単純になっている分，通信が早い．

TCP を使うプログラムを書くときに，OS が提供する通信チャンネルをプログラム上に表す必要がある．そこで使われる概念はソケットである．新しく作られたソケットは何もできないが，そのソケットを通信チャンネルに束縛すれば，そのソケットに書くときはチャンネルへの送信，読むときはチャンネルからの受信という意味になる．

ソケットに関するいくつかのシステムコールを見よう．

- *val socket : socket_domain -> socket_type -> int -> file_descr*
  socket domain typ proto は新しいソケットを作る．ファイルと同様に，ハンドルが返される．引数は，接続性を表す domain，TCP か UDP の別を表す type があるが，インターネット用の TCP ソケットなら，socket PF_INET SOCK_STREAM 0 で作る．

```
type socket_domain =
    PF_UNIX                     (* Unix domain *)
  | PF_INET                     (* Internet domain (IPv4) *)
  | PF_INET6                    (* Internet domain (IPv6) *)
type socket_type =
    SOCK_STREAM                 (* Stream socket *)
  | SOCK_DGRAM                  (* Datagram socket *)
  | SOCK_RAW                    (* Raw socket *)
  | SOCK_SEQPACKET              (* Sequenced packets socket *)
```

- *val bind : file_descr -> sockaddr -> unit*

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

bind sock addr はあるソケットをあるポート番号に束縛する．今後のこのポート番号への接続依頼はこのソケットに届けられる．第 1 引数はソケットハンドル，第 2 引数は束縛するアドレスとポート番号．接続の種類によって，具体的な sockaddr が変わる．sock の domain と同じ種類を使わなければならない．インターネットだと，ADDR_INET である．int はポート番号．inet_addr は束縛されるアドレス．通常 inet_addr_any で，このコンピュータが持っている全てのアドレスを使うが，それ以外の IP アドレスを与えると，そのアドレスに来た接続依頼だけが対象となる．例えば，ループバックアドレス inet_addr_loopback にすることで，他のコンピュータからの接続依頼を不可にできる．

- *val listen : file_descr -> int -> unit*
  既に bind sock n されたソケットに対して，TCP の接続を受け付けるようにする．第 2 引数は接続待ちの依頼を何個まで許すかを決める．

- *val accept : file_descr -> file_descr * sockaddr*
  accept sock は接続待ちの依頼を一個開く．返り値はその接続を扱うための新しいソケットハンドルおよび相手の IP アドレスとポート番号である．

- *val connect : file_descr -> sockaddr -> unit* connect sock addr はあるアドレスに接続を依頼する．accept されたら，渡されたソケット sock はこの接続へのハンドルになる．

- *val read : file_descr -> string -> int -> int -> int*
  val write : file_descr -> string -> int -> int -> int
  val close : file_descr -> unit
  ソケットだけでなく，一般的なファイルに対しても使える．
  read sock buffer pos n は sock から n 個以下の文字を読み，文字列 buffer の pos 以降に書き込む．buffer の長さは $pos + n$ 以上でなければならない．結果は実際に読めて文字の数 (1 以上)
  write sock buffer pos n は sock に対して，文字列 buffer の pos 以降の n 個の文字を書こうとする．buffer の長さは $pos + n$ 以上でなければならない．結果は実際に書けた文字の数 (1 以上)
  close sock はソケットを閉じる．閉じられないソケットのアドレスは再利用できないので要注意．

プログラム例

とても単純なプログラム例である．サーバはあるポート番号で待ち，接続が来たらそこから一行だけ読み，それを端末に出力する．クライアントは一行を端末から読み，サーバに送る．

実習 まずサーバを書いて，ocamlc unix.cma server.ml -o server でコンパイルする．移すときは，細かい間違いに気を付けなければならない．

server ができたら，それを実行する．別の端末で

```
$ telnet localhost 6333
```

でテストができる．exit を送るとサーバが終了する．

クライアントもコンパイルして，そちらでも使って見る．

時間があれば，サーバに今までの全メッセージを記録するようにし，クライアントから log という文字列を送ったら，その全記録を送り返すようにする．

サーバ (server.ml)

```
open Unix                                                    (* Unix の型定義を使う *)

let bufsize = 80                                             (* バッファーの長さ *)
let port = 6333                                              (* ポート番号 *)

let main () =
  let s = Unix.socket PF_INET SOCK_STREAM 0 in
  Unix.setsockopt s SO_REUSEADDR true;                       (* close し忘れたときのために… *)
  let server = ADDR_INET(inet_addr_any, port) in
  Unix.bind s server;                                        (* 上記ポート番号で待機 *)
  Unix.listen s 1;                                           (* 待機接続を一個だけ許す *)

  let rs, client = Unix.accept s in                          (* 接続を受け付ける *)
  let n = ref 0 and buf = String.create bufsize in
  while n := Unix.read rs buf 0 bufsize; !n > 0 do           (* 切られるまで繰り返す *)
    for i = 0 to !n-1 do
      buf.[i] <- Char.uppercase buf.[i]                      (* 小文字を大文字に *)
    done;
    ignore (Unix.write rs buf 0 !n)                          (* 全部書けない場合を無視! *)
  done;
  close rs;                                                  (* 全てのソケットを閉じる *)
  close s

let () =
  try main ()
  with Unix_error (err, f, s) ->                             (* エラーを報告する *)
    Printf.eprintf "Unix error in %s: %s (%s)\n%!"
      f (Unix.error_message err) s
```

クライアント (client.ml)

```
open Unix

let bufsize = 80
let port = 6333

let main () =
  let s = Unix.socket PF_INET SOCK_STREAM 0 in
  let addr = ADDR_INET(inet_addr_loopback, port) in

  Unix.connect s addr;                                       (* localhost に同じポート番号で接続する *)

  let buf = String.create bufsize in
  while
    let inp = read_line () in                                (* 一行読んで，サーバに送る *)
    let len = String.length inp in
    len <> 0 && Unix.write s inp 0 len = len                 (* 全部書けなければ終わり?! *)
  do
    let len = Unix.read s buf 0 bufsize in                   (* サーバからの返事を待つ *)
    print_endline (String.sub buf 0 len)
  done;
  Unix.close s                                               (* ソケットを閉じる *)

let () =
  try main ()
  with Unix_error (err, f, s) ->
    Printf.eprintf "Unix error in %s: %s (%s)\n%!"
      f (Unix.error_message err) s
```