Reading: Abelson & Sussman, Section 3.4

• Client/server programming paradigm

Before networks, most programs ran on a single computer. Today it's common for programs to involve cooperation between computers. The usual reason is that you want to run a program on your computer that uses data located elsewhere. A common example is using a browser on your computer to read a web page stored somewhere else.

To make this cooperation possible, *two* programs are actually required: the *client* program on your personal computer and the *server* program on the remote computer. Sometimes the client and the server are written by a single group, but often someone publishes a *standard* document that allows any client to work with any server that follows the same standard. For example, you can use Mozilla, Netscape, or Internet Explorer to read most web pages, because they all follow standards set by the World Wide Web Consortium.

For this course we provide a sample client/server system, implementing a simple Instant Message protocol. The files are available in

```
~cs61a/lib/im-client.scm
~cs61a/lib/im-server.scm
```

To use them, you must first start a server. Load `im-server.scm` and call the procedure `im-server-start`; it will print the IP (Internet Protocol) address of the machine you're using, along with another number, the *port* assigned to the server. Clients will use these numbers to connect to the server. Port numbers are important because there might be more than one server program running on the same computer, and also to keep track of connections from more than one client.

(Why don't you need these numbers when using "real" network software? You don't need to know the IP address because your client software knows how to connect to *nameservers* to translate the host names you give it into addresses. And most client/server protocols use fixed, *registered* port numbers that are built into the software. For example, web browsers use port 80, while the `ssh` protocol you may use to connect to your class account from home uses port 22. But our sample client/server protocol doesn't have a registered port number, so the operating system assigns a port to the server when you start it.)

To connect to the server, load `im-client.scm` and call `im-enroll` with the IP address and port number as arguments. (Details are in this week's lab assignment.) Then use the `im` procedure to send a message to other people connected to the same server.

This simple implementation uses the Scheme interpreter as its user interface; you send messages by typing Scheme expressions. Commercial Instant Message clients have a more ornate user interface, that accept mouse clicks in windows listing other clients to specify the recipient of a message. But our version is realistic in the way it uses the network; the IM client on your home computer connects to a particular port on a particular server in order to use the facility. (The only difference is that a large commercial IM system will have more than one server; your client connects to the one nearest you, and the servers send messages among themselves to give the illusion of one big server to which everyone is connected.)

In the news these days, client/server protocols are sometimes contrasted with another approach called *peer-to-peer* networking, such as file-sharing systems like Napster and Kazaa. The distinction is social rather than strictly technical. In each individual transaction using a peer-to-peer protocol, one machine is acting as a server and the other as a client. What makes it peer-to-peer networking is that any machine using the protocol can play either role, unlike the more usual commercial networking idea in which rich companies operate servers and ordinary people operate clients.

Internet primitives in STk

STk defines sockets, on systems which support them, as first class objects. Sockets permit processes to communicate even if they are on different machines. Sockets are useful for creating client-server applications.

(make-client-socket hostname port-number)

`make-client-socket` returns a new socket object. This socket establishes a link between the running application listening on port `port-number` of `hostname`.

(socket? socket)

Returns `#t` if `socket` is a socket, otherwise returns `#f`.

(socket-host-name socket)

Returns a string which contains the name of the distant host attached to `socket`. If `socket` was created with `make-client-socket`, this procedure returns the official name of the distant machine used for connection. If `socket` was created with `make-server-socket`, this function returns the official name of the client connected to the socket. If no client has yet used the socket, this function returns `#f`.

(socket-host-address socket)

Returns a string which contains the IP number of the distant host attached to `socket`. If `socket` was created with `make-client-socket`, this procedure returns the IP number of the distant machine used for connection. If `socket` was created with `make-server-socket`, this function returns the address of the client connected to the socket. If no client has yet used the socket, this function returns `#f`.

(socket-local-address socket)

Returns a string which contains the IP number of the local host attached to `socket`.

(socket-port-number socket)

Returns the integer number of the port used for `socket`.

(socket-input socket)
(socket-output socket)

Returns the port associated for reading or writing with the program connected with `socket`. If no connection has been established, these functions return `#f`. The following example shows how to make a client socket. Here we create a socket on port 13 of the machine `kaolin.unice.fr`. [Port 13 is generally used for testing: making a connection to it returns the distant system's idea of the time of day.]

```
(let ((s (make-client-socket "kaolin.unice.fr" 13)))
  (format #t "Time is: ~A\n" (read-line (socket-input s)))
  (socket-shutdown s))
```

(make-server-socket)
(make-server-socket port-number)

`make-server-socket` returns a new socket object. If `port-number` is specified, the socket listens on the specified port; otherwise, the communication port is chosen by the system.

(socket-accept-connection socket)

`socket-accept-connection` waits for a client connection on the given socket. If no client is already waiting for a connection, this procedure blocks its caller; otherwise, the first connection request on the queue of pending connections is connected to socket. This procedure must be called on a server socket created with `make-server-socket`. The return value of `socket-accept-connection` is undefined. The following example is a simple server which waits for a connection on the port 1234. Once the connection with the distant program is established, we read a line on the input port associated to the socket and we write the length of this line on its output port. [Under Unix, you can simply connect to listening socket with the telnet command. With the given example, this can be achieved by typing the command

```
telnet localhost 1234
```

in a shell window.]

```
(let ((s (make-server-socket 1234)))
  (socket-accept-connection s)
  (let ((l (read-line (socket-input s))))
    (format (socket-output s) "Length is: ~A\n" (string-length l))
    (flush (socket-output s)))
  (socket-shutdown s))
```

(socket-shutdown socket)
(socket-shutdown socket close)

`Socket-shutdown` shuts down the connection associated to `socket`. `Close` is a boolean; it indicates if the socket must be closed or not, when the connection is destroyed. Closing the socket forbids further connections on the same port with the `socket-accept-connection` procedure. Omitting a value for `close` implies closing the socket. The return value of `socket-shutdown` is undefined. The following example shows a simple server: when there is a new connection on the port number 1234, the server displays the first line sent to it by the client, discards the others and goes back waiting for further client connections.

```
(let ((s (make-server-socket 1234)))
  (let loop ()
    (socket-accept-connections)
    (format #t "I've read: ~A\n" (read-line (socket-input s)))
    (socket-shutdown s #f)
    (loop)))
```

(socket-down? socket)

Returns `#t` if socket has been previously closed with socket-shutdown. It returns `#f` otherwise.

(socket-dup socket)

Returns a copy of `socket`. The original and the copy socket can be used interchangeably. However, if a new connection is accepted on one socket, the characters exchanged on this socket are not visible on the other socket. Duplicating a socket is useful when a server must accept multiple simultaneous connections. The following example creates a server listening on port 1234. This server is duplicated and, once two clients are present, a message is sent on both connections.

```
(define s1 (make-server-socket 1234))
(define s2 (socket-dup s1))
(socket-accept-connection s1)
(socket-accept-connection s2) ;; blocks until two clients are present
(display "Hello,\n" (socket-output s1))
(display "world\n" (socket-output s2))
(flush (socket-output s1))
(flush (socket-output s2))
```

(when-socket-ready socket handler)
(when-socket-ready socket)

Defines a handler for `socket`. The handler is a thunk which is executed when a connection is available on socket. If the special value `#f` is provided as `handler`, the current handler for socket is deleted. If a handler is provided, the value returned by `when-socket-ready` is undefined. Otherwise, it returns the handler currently associated to `socket`. This procedure, in conjunction with `socket-dup`, permits building multiple-client servers which work asynchronously. Such a server is shown below.

```
(define p (make-server-socket 1234))
(when-socket-ready p
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      (register-connection (socket-dup p) count))))
(define register-connection
  (let ((sockets '()))
    (lambda (s cnt)
      ;; Accept connection
      (socket-accept-connection s)
      ;; Save socket somewhere to avoid GC problems
      (set! sockets (cons s sockets))
      ;; Create a handler for reading inputs from this new connection
      (let ((in (socket-input s))
            (out (socket-output s)))
        (when-port-readable in
          (lambda ()
            (let ((l (read-line in)))
              (if (eof-object? l)
                  ;; delete current handler
                  (when-port-readable in #f)
                  ;; Just write the line read on the socket
                  (begin (format out "On #~A --> ~A\n" cnt l)
                         (flush out)))))))))
```