# Networking in PLT Scheme

Shriram Krishnamurthi

Dec 20, 2000

PLT Scheme has simple yet powerful primitives for establishing network connections. Learn more about them by searching on"TCP/IP" in Help Desk.

We will build a few applications in this tutorial, starting with something mind-numbingly simple, and growing to something that begins to approach a useful application. [NOTE: Tutorial hasn't grown all the way out that far yet.]

## 1 The `Ping-Pong Duet`

The first application is a client-server combination of staggering simplicity. The client issues requests that consist of the symbol `'ping`. The server, upon receipt of this request, responds with the symbol `'pong`. One time. The client prints the server's response. That's all.

First, the two programs must agree on a common port where the server will listen so the client can connect.

```
(define SERVICE-PORT 2000)
(define SERVER-HOST "localhost")
```

To test your programs, pick a "random" port number between 1025 and 65535. The bigger numbers are more barren, so you are less likely to interfere with another service that already exists on your machine.

The client is easier to write. It simply uses `TCP-CONNECT` to establish a connection to the server.

```
(define (client)
  (let-values ([(server->me me->server)
                (tcp-connect SERVER-HOST SER\
VICE-PORT)])
    (write 'ping me->server)
    (close-output-port me->server)
    (let ([response (read server->me)])
      (display response) (newline)
      (close-input-port server->me))))
```

This defines `CLIENT` as a procedure of no arguments (so the body doesn't evaluate until we invoke the procedure). `TCP-CONNECT` returns two values (look up "multiple return values" in Help Desk). The first is an input port, to which the server writes data, and the second an output port, from which the server reads data. The naming convention used above helps me keep them straight. The `WRITE` statement writes `'ping` to the port being read by the server. Having written the message, the client closes its ports and exits.

In the above example, we assume both client and server reside on the same machine (hence the use of the hostname `"localhost"`). The client can reside on an entirely different machine, however.

The server's definition is slightly more complex. Here's the server:

```
(define (server)
  (let ([listener (tcp-listen SERVICE-PORT)])
    (let-values ([(client->me me->client)
                  (tcp-accept listener)])
 (if (eq? (read client->me) 'ping)
           (write 'pong me->client)
           (write 'who-are-you? me->client))
       (close-output-port me->client)
       (close-input-port client->me))))
```

The server must first create a —verb—"listener"—. The listener is woken up when a network connection comes in on the chosen port. `TCP-ACCEPT` accepts responses queued at the server.

If we combine these three code fragments (constants, client

and server) and run them in a single Scheme session, we ... can't. There's a problem.

If we run the client first, it tries to connect with the server, which isn't yet running, and we get an error saying there's no response from the common port.

If we run the server first, it creates a listener, then executes the `TCP-ACCEPT` expression. This blocks on a request before it can continue. But we need it to return control to the prompt so we can start the client.

In short, we can't run either one first.

There are three ways out of this jam.

First, we use two separate copies of Scheme (i.e., separate processes). The first process runs the server. The second one runs the client. Note that each process must have the definition of its procedure and the constant definitions. When run after starting the server, the client will return the value pong.

Second, we can just run client and server on different machines. This is really just a special case of the first solution, but it also lets you experiment with connecting to different machines. To do this, you'd have to edit the value associated with `SERVER-HOST` to be the name of the machine running the server.

Third, we can use threads. (Read up about `"threads"` in Help Desk.) We can thus invoke both the client and server in the same Scheme process by running

```
> (load )
> (thread server)
>     [back to the Scheme prompt; server runs\
in separate thread]
```

`THREAD` expects a procedure of no arguments as its first argument, which is exactly what `SERVER` is.

```
> (client)
pong
```

In both cases, SERVER exits as soon as it has serviced its request. If you invoke SERVER with THREAD you won't notice this. If you run the client and server in two separate processes, you will.

That concludes our first example.

## 2 Queueing up for Tokens

The server and client in the first example ran only once. Typical servers run forever, accepting and servicing requests as they arrive. We'll create such a server as our second example.

We will once again build a client-server combination. The server generates token numbers, much like the machines that issue serial numbers to people in queues. The server initializes at zero, and each request generates the next token number. The client is a function that consumes one argument, which is the number of tokens to receive. It contacts the server as many times as specified in its argument, and returns a list of the resulting tokens.

If we were writing this program without networking, it would look as follows:

```
  (define serve-next-token
    (let ([next-token -1])
;; so first value return is 0
      (lambda ()
 (set! next-token (+ next-token 1))
 next-token)))

  (define token-client
    (lambda (how-many-tokens)
      (if (<= how-many-tokens 0)
   '()
   (cons (serve-next-token)
  (token-client (- how-many-tokens 1))))))
```

This program behaves as follows:

```
  (token-client 5)  ==>  '(0 1 2 3 4)
  (token-client 3)  ==>  '(5 6 7)
```

Make sure you understand this code before proceeding.

First, establish the server's locus:

```
  (define SERVICE-PORT 2005)
  (define SERVER-HOST "localhost")
```

The client is again pretty simple:

```
(define token-client
  (lambda (how-many-tokens)
    (if (<= how-many-tokens 0)
 '()
 (let-values ([(server->me me->server)
 (tcp-connect SERVER-HOST SERVICE-PORT)])
   (let ([token (read server->me)])
     (close-input-port server->me)
     (close-output-port me->server)
     (cons token
    (token-client (- how-many-tokens 1))))))))
```

The server's basic structure looks the same:

```
(define (server)
  (let ([listener (tcp-listen SERVICE-PORT)])
    ...))
```

except it must do two things: (1) keep track of the last token number, and loop to handle multiple requests. Thus:

```
(define server
  (let ([next-token -1])
    (lambda ()
(let ([listener (tcp-listen SERVICE-PORT)])
        (let loop ()
    (let-values ([(client->me me->client)
    (tcp-accept listener)])
             (set! next-token (+ next-token 1))
      (close-input-port client->me)
      (write next-token me->client)
      (close-output-port me->client))
           (loop))))))
```

Note that the server does *not* need to create multiple listeners. It creates the listener for that service just once. It accepts connections from the listener multiple times. Now, assuming the server is running (either in a separate process, on a separate machine, or in its own thread), running the client returns the expected values:

```
> (token-client 5)
(0 1 2 3 4)
> (token-client 3)
(5 6 7)
```

Note the very subtle yet critical difference between the server above and this one:

```
(define server
  (let ([next-token -1])
    (lambda ()
      (let loop ()
;; order of these
   (let ([listener (tcp-listen SERVICE-PORT)])
;; 2 lines swapped
     (let-values ([(client->me me->client)
     (tcp-accept listener)])
              (set! next-token (+ next-token 1))
       (close-input-port client->me)
       (write next-token me->client)
       (close-output-port me->client))
            (loop))))))
```

The above server is *buggy*! Each time through the loop it tries to create a new listener. The first time it succeeds; on the second attempt, the invocation of `TCP-LISTEN` fails because there is already a listener on that port — created by this very server!

# 3 Variations on the Token Server: Reusing a Connection and Obtaining Consecutive Numbers

There are potentially two problems with the token server above. We address both these problems in this section.

First, the sample interactions with the token server before this section suggest that the tokens will always be consecutive, as they are in the sequential world. In fact, however, the server lives in a concurrent universe. Each time through the loop, `TOKEN-CLIENT` establishes a *fresh* connection with the server. Besides being somewhat inefficient, this also means that a different client may connect between two consecutive connections by your client, and may thus grab one or more of the intermediate numbers. So you might see an interaction like

```
> (token-client 5)
(0 1 3 4 6)
```

(You probably won't for small numbers of tokens, but if you set off two processes each requesting a large number of tokens — say 1,000 each — from the same server, you ought to find that they aren't all consecutive.)

A related problem is that the client connects to the server each time through its loop. This wastes network resources by repeatedly re-establishing connections. A better solution is for the client and server to have a "dialog": once connected, the server keeps providing the client with tokens until the client has exhausted its demand.

In this rewrite, the client sends the server one of two messages: `'more`, as long as it needs more numbers, and `'enough`, when it no longer needs any more numbers. The server responds appropriately.

```
  (define token-client
    (lambda (how-many-tokens)
      (let-values ([(server->me me->server)
            (tcp-connect SERVER-HOST SERVICE-\
PORT)])
        (let loop ([how-many-more how-many-tok\
ens])
          (if (<= how-many-more 0)
            (begin
              (close-input-port server->me)
              (write 'enough me->server)
              (close-output-port me->server)
              '())
            (begin
              (write 'more me->server)
              (newline me->server)
              (flush-output me->server)
              (cons (read server->me)
                    (loop (- how-many-more 1))\
)))))))

  (define server
    (let ([next-token -1])
      (lambda ()
 (let ([listener (tcp-listen SERVICE-PORT)])
          (let server-loop ()
     (let-values ([(client->me me->client)
     (tcp-accept listener)])
              (let per-client-loop ()
                (let ([request (read client->m\
e)])
                  (case request
                    [(enough)
                     (close-input-port client->m\
e)
                     (close-output-port me->client)
                            (server-loop)]
                           [(more)
                            (set! next-token (+ next-\
token 1))
                            (write next-token me->cli\
ent)
                            (newline me->client)
                     (flush-output me->client)
                            (per-client-loop)])))))))\
))
```

The `NEWLINE` and `FLUSH-OUTPUT` are necessary to make the buffers to be flushed to the network device and transmitted.

This program also guarantees that the tokens will be consecutive, because the server does not service a new client until it is done responding to the current one.

Another way to make more effective use of a connection and to ensure consecutive tokens is to have your server return several (consecutive) tokens at once. How to package up the tokens? All our examples thusfar have transmitted only symbols and numbers. In fact, however, as with I/O operations on any other port, you may use any readable and writeable datum [NOTE: insert Help Desk reference here]. As a simple example, here's a rewrite of the token server and client where the client simply informs the server of how many tokens it needs, and the server returns a list of that many tokens.

```
  (define token-client
    (lambda (how-many-tokens)
      (let-values ([(server->me me->server)
            (tcp-connect SERVER-HOST SERVICE-\
PORT)])
 (write how-many-tokens me->server)
        (newline me->server)
        (flush-output me->server)
        (let ([tokens (read server->me)])
          (close-input-port server->me)
          (close-output-port me->server)
          tokens))))

  (define server
    (let ([next-token -1])
      (lambda ()
 (let ([listener (tcp-listen SERVICE-PORT)])
```

```
          (let server-loop ()
     (let-values ([(client->me me->client)
     (tcp-accept listener)])
               (write
  (let count-loop ([how-many-more (read client\
->me)])
    (if (<= how-many-more 0)
        '()
        (begin
   (set! next-token (+ next-token 1))
   (cons next-token
         (count-loop (- how-many-more 1))))))
                me->client)
               (close-output-port me->client)
               (close-input-port client->me)
               (server-loop)))))))
```

Notice that the first argument to `WRITE` is a loop that computes a sequence of tokens.

Both servers in this section come at a price. One long request can tie this server down and make it unable to service other requests. Eventually, the number of pending requests may become too large and new client requests may be denied. (See the second parameter to `TCP-LISTEN`.)

That concludes our short tour of a `ping-pong` server.

**About the Author** Shriram Krishnamurthi is Assistant Professor of Computer Science at Brown University. He researches the semantics, verification and implementation of advanced programming languages. He is particularly interested in the problems of dynamically extensible software systems. He is a co-author of the **DrScheme** programming environment, the FASTLINK genetic linkage analysis package, and the book *How to Design Programs* (MIT Press, 2001). He and his colleagues are now writing *How to Use Scheme.* He is also the webmaster of the famous web site *http://www.schemers.org.* He also coordinates the *TeachScheme!* high school computer science outreach program. He can be reached by email: `sk@cs.brown.edu`

---

[**Microsoft to Clobber Linux**] Byron Acohido, a journalist of **USA TODAY** reported at Seattle that "Microsoft is escalating its war against Linux, the free operating system begun as a hobby of Internet dabblers but increasingly the darling of bankers, retailers and Hollywood special effects wizards".

According to Dec 26, 2001 memo leaked to technology Web site *theregister.uk.com*, Microsoft Senior Vice President Brian Valentine urges his Windows sales force to "modify" traditional approaches and "dig deeper" to find out where companies are using Linux. He assures his troops they will soon get "independent" studies and spreadsheet tools useful for obliterating "the perception that Linux is free."

Microsoft will probably try to show that Linux is costly to maintain and service over the long term, experts say. "We're working hard to debunk myths around Linux," he writes. "We're approaching this in waves."

Microsoft declines to comment on Valentine's call to arms. The software giant considers his rallying cry simply being proactive in the marketplace, says a source familiar with the matter. But the memo has been a hot topic in industry circles, and some observers say it could ultimately backfire.

Unlike others Microsoft has vanquished, Linux is more a grass-roots religion than a vulnerable corporate target. While Microsoft jealously guards its core software code, Linux code is open and continually improved by thousands of top programmers around the world who believe software should benefit society.

...

Some think Microsoft should likewise embrace Linux. But Valentine's memo suggests Microsoft is resorting to old tactics at the risk of giving Linux a bigger forum to tout its success stories, says Dan Kusnetzky, an analyst with research firm IDC.

...

Digested from *http://www.usatoday.com/money/tech/2002-01-04-microsoft.htm* —**FSM**