More: Systems Programming with PLT Scheme

Version 4.0.1

June 22, 2008

In contrast to the impression that §“**Quick**: An Introduction to PLT Scheme with Pictures” may give, PLT Scheme is not just another pretty face. Underneath the graphical facade of DrScheme lies a sophisticated toolbox for managing threads and processes, which is the subject of this tutorial.

Specifically, we show how to build a secure, multi-threaded, servlet-extensible, continuation-based web server. We use much more of the language than in §“**Quick**: An Introduction to PLT Scheme with Pictures”, and we expect you to click on syntax or function names that you don’t recognize (which will take you to the relevant documentation). Beware that the last couple of sections present material that is normally considered difficult. If you’re still new to Scheme and have relatively little programming experience, you may want to skip to §“**Guide**: PLT Scheme”.

To get into the spirit of this tutorial, we suggest that you set DrScheme aside for a moment, and switch to raw `mzscheme` in a terminal. You’ll also need a text editor, such as `emacs` or `vi`. Finally, you’ll need a web client, perhaps `lynx` or `firefox`.

Of course, if you’re already spoiled, you can keep using DrScheme.

1 Ready...

Download PLT Scheme, install, and then start `mzscheme` with no command-line arguments:

```
$ mzscheme
Welcome to MzScheme
>
```

If you're using a plain terminal, if you have GNU Readline installed on your system, and if you'd like Readline support in `mzscheme`, then evaluate `(require readline)`. If you also evaluate `(install-readline!)`, then your `"~/.mzschemerc"` is updated to load Readline whenever you start `mzscheme` for interactive evaluation.

Unfortunately, for legal reasons related to GPL vs. LGPL, `mzscheme` cannot provide Readline automatically.

```
> (require readline)
> (install-readline!)
```

2 Set...

In the same directory where you started `mzscheme`, create a text file `"serve.ss"`, and start it like this:

```
#lang scheme

(define (go)
  'yep-it-works)
```

Here's the whole program so far in plain text: step 0.

3 Go!

Back in `mzscheme`, try loading the file and running `go`:

```
> (enter! "serve.ss")
 [loading serve.ss]
> (go)
yep-it-works
```

Try modifying `"serve.ss"`, and then running `(enter! "serve.ss")` again to re-load the module, then check your changes.

4 “Hello World” Server

We’ll implement the web server through a `serve` function that takes a IP port number for client connections:

```
(define (serve port-no)
  ...)
```

The server accepts TCP connections through a *listener*, which we create with `tcp-listen`. To make interactive development easier, we supply `#t` as the third argument to `tcp-listen`, which lets us re-use the port number without waiting on TCP timeouts.

```
(define (serve port-no)
  (define listener (tcp-listen port-no 5 #t))
  ...)
```

The server must loop to accept connections from the listener:

```
(define (serve port-no)
  (define listener (tcp-listen port-no 5 #t))
  (define (loop)
    (accept-and-handle listener)
    (loop))
  (loop))
```

Our `accept-and-handle` function accepts a connection using `tcp-accept`, which returns two values: a stream for input from the client, and a stream for output to the client.

```
(define (accept-and-handle listener)
  (define-values (in out) (tcp-accept listener))
  (handle in out)
  (close-input-port in)
  (close-output-port out))
```

To handle a connection, for now, we’ll read and discard the request header, and then write a “Hello, world!” web page as the result:

```
(define (handle in out)
  ; Discard the request header (up to blank line):
  (regexp-match #rx"(\r\n|^)\r\n" in)
  ; Send reply:
  (display "HTTP/1.0 200 Okay\r\n" out)
  (display "Server: k\r\nContent-Type: text/html\r\n\r\n" out)
  (display "<html><body>Hello, world!</body></html>" out))
```

Note that `regexp-match` operates directly on the input stream, which is easier than bothering with individual lines.

Here’s the whole program so far in plain text: step 1.

Copy the above three definitions—`serve`, `accept-and-handle`, and `handle`—into `"serve.ss"` and re-load:

```
> (enter! "serve.ss")
 [re-loading serve.ss]
> (serve 8080)
```

Now point your browser to `http://localhost:8080` (assuming that you used `8080` as the port number, and that the browser is running on the same machine) to receive a friendly greeting from your web server.

5 Server Thread

Before we can make the web server respond in more interesting ways, we need to get a Scheme prompt back. Typing Ctl-C in your terminal window interrupts the server loop:

In DrScheme, instead of typing Ctl-C, click the Stop button once.

```
> (serve 8080)
^Cuser break
>
```

Unfortunately, we cannot now re-start the server with the same port number:

```
> (serve 8080)
tcp-listen: listen on 8080 failed (address already in use)
```

The problem is that the listener that we created with `serve` is still listening on the original port number.

To avoid this problem, let's put the listener loop in its own thread, and have `serve` return immediately. Furthermore, we'll have `serve` return a function that can be used to shut down the server thread and TCP listener:

```
(define (serve port-no)
  (define listener (tcp-listen port-no 5 #t))
  (define (loop)
    (accept-and-handle listener)
    (loop))
  (define t (thread loop))
  (lambda ()
    (kill-thread t)
    (tcp-close listener)))
```

Here's the whole program so far in plain text: step 2.

Try the new one:

```
> (enter! "serve.ss")
[re-loading serve.ss]
> (define stop (serve 8081))
```

Your server should now respond to `http://localhost:8081`, but you can shut down and restart the server on the same port number as often as you like:

```
> (stop)
> (define stop (serve 8081))
> (stop)
> (define stop (serve 8081))
> (stop)
```

6 Connection Threads

In the same way that we put the main server loop into a background thread, we can put each individual connection into its own thread:

```
(define (accept-and-handle listener)
  (define-values (in out) (tcp-accept listener))
  (thread
   (lambda ()
     (handle in out)
     (close-input-port in)
     (close-output-port out))))
```

Here's the whole program so far in plain text: step 3.

With this change, our server can now handle multiple threads at once. The handler is so fast that this fact will be difficult to detect, however, so try inserting `(sleep (random 10))` before the `handle` call above. If you make multiple connections from the web browser at roughly the same time, some will return soon, and some will take up to 10 seconds. The random delays will be independent of the order in which you started the connections.

7 Terminating Connections

A malicious client could connect to our web server and not send the HTTP header, in which case a connection thread will idle forever, waiting for the end of the header. To avoid this possibility, we'd like to implement a timeout for each connection thread.

One way to implement the timeout is to create a second thread that waits for 10 seconds, and then kills the thread that calls `handle`. Threads are lightweight enough in Scheme that this watcher-thread strategy works well:

```
(define (accept-and-handle listener)
  (define-values (in out) (tcp-accept listener))
  (define t (thread
             (lambda ()
               (handle in out)
               (close-input-port in)
               (close-output-port out))))
  ; Watcher thread:
  (thread (lambda ()
            (sleep 10)
            (kill-thread t))))
```

This first attempt isn't quite right, because when the thread is killed, its `in` and `out` streams remain open. We could add code to the watcher thread to close the streams as well as kill the thread, but Scheme offers a more general shutdown mechanism: *custodians*. A custodian is a kind of container for all resources other than memory, and it supports a `custodian-shutdown-all` operation that terminates and closes all resources within the container, whether they're threads, streams, or other kinds of limited resources.

Whenever a thread or stream is created, it is placed into the current custodian as determined by the `current-custodian` parameter. To place everything about a connection into a custodian, we `parameterize` all the resource creations to go into a new custodian:

```
(define (accept-and-handle listener)
  (define cust (make-custodian))
  (parameterize ([current-custodian cust])
    (define-values (in out) (tcp-accept listener))
    (thread (lambda ()
              (handle in out)
              (close-input-port in)
              (close-output-port out))))
  ; Watcher thread:
  (thread (lambda ()
            (sleep 10)
            (custodian-shutdown-all cust))))
```

With this implementation, `in`, `out`, and the thread that calls `handle` all belong to `cust`. In addition, if we later change `handle` so that it, say, opens a file, then the file handles will also belong to `cust`, so they will be reliably closed when `cust` is shut down.

In fact, it's a good idea to change `serve` to that it uses a custodian, too:

```
(define (serve port-no)
  (define main-cust (make-custodian))
  (parameterize ([current-custodian main-cust])
    (define listener (tcp-listen port-no 5 #t))
    (define (loop)
      (accept-and-handle listener)
      (loop))
    (thread loop))
  (lambda ()
    (custodian-shutdown-all main-cust)))
```

That way, the `main-cust` created in `serve` not only owns the TCP listener and the main server thread, it also owns every custodian created for a connection. Consequently, the revised shutdown procedure for the server immediately terminates all active connections, in addition to the main server loop.

Here's the whole program so far in plain text: step 4.

After updating the `serve` and `accept-and-handle` functions as above, here's how you can simulate a malicious client:

```
> (enter! "serve.ss")
 [re-loading serve.ss]
> (define stop (serve 8081))
> (define-values (cin cout) (tcp-connect "localhost" 8081))
```

Now wait 10 seconds. If you try reading from `cin`, which is the stream that sends data from the server back to the client, you'll find that the server has shut down the connection:

```
> (read-line cin)
#<eof>
```

Alternatively, you don't have to wait 10 seconds if you explicitly shut down the server:

```
> (define-values (cin2 cout2) (tcp-connect "localhost" 8081))
> (stop)
> (read-line cin2)
#<eof>
```

8 Dispatching

It's finally time to generalize our server's "Hello, World!" response to something more useful. Let's adjust the server so that we can plug in dispatching functions to handle requests to different URLs.

To parse the incoming URL and to more easily format HTML output, we'll require two extra libraries:

```
(require net/url xml)
```

The `xml` library gives us `xexpr->string`, which takes a Scheme value that looks like HTML and turns it into actual HTML:

```
> (xexpr->string '(html (head (title "Hello")) (body "Hi!")))
"<html><head><title>Hello</title></head><body>Hi!</body></html>"
```

We'll assume that our new `dispatch` function (to be written) takes a requested URL and produces a result value suitable to use with `xexpr->string` to send back to the client:

```
(define (handle in out)
  (define req
    ; Match the first line to extract the request:
    (regexp-match #rx"^GET (.+) HTTP/[0-9]+\\.[0-9]+"
                  (read-line in)))
  (when req
    ; Discard the rest of the header (up to blank line):
    (regexp-match #rx"(\r\n|^)\r\n" in)
    ; Dispatch:
    (let ([xexpr (dispatch (list-ref req 1))])
      ; Send reply:
      (display "HTTP/1.0 200 Okay\r\n" out)
      (display "Server: k\r\nContent-Type: text/html\r\n\r\n" out)
      (display (xexpr->string xexpr) out))))
```

The `net/url` library gives us `string->url`, `url-path`, `path/param-path`, and `url-query` for getting from a string to parts of the URL that it represents:

```
> (define u (string->url "http://localhost:8080/foo/bar?x=bye"))
> (url-path u)
(#<path/param> #<path/param>)
> (map path/param-path (url-path u))
("foo" "bar")
> (url-query u)
((x . "bye"))
```

We use these pieces to implement `dispatch`. The `dispatch` function consults a hash table

that maps an initial path element, like `"foo"`, to a handler function:

```
(define (dispatch str-path)
  ; Parse the request as a URL:
  (define url (string->url str-path))
  ; Extract the path part:
  (define path (map path/param-path (url-path url)))
  ; Find a handler based on the path's first element:
  (define h (hash-ref dispatch-table (car path) #f))
  (if h
      ; Call a handler:
      (h (url-query url))
      ; No handler found:
      `(html (head (title "Error"))
             (body
              (font ((color "red"))
                    "Unknown page: "
                    ,str-path)))))

(define dispatch-table (make-hash))
```

With the new `require` import and new `handle`, `dispatch`, and `dispatch-table` definitions, our "Hello World!" server has turned into an error server. You don't have to stop the server to try it out. After modifying `"serve.ss"` with the new pieces, evaluate `(enter! "serve.ss")` and then try again to connect to the server. The web browser should show an "Unknown page" error in red.

We can register a handler for the `"hello"` path like this:

```
(hash-set! dispatch-table "hello"
           (lambda (query)
             `(html (body "Hello, World!")))))
```

Here's the whole program so far in plain text: step 5.

After adding these lines and evaluating `(enter! "serve.ss")`, opening `http://localhost:8081/hello` should produce the old greeting.

9 Servlets and Sessions

Using the `query` argument that is passed to a handler by `dispatch`, a handler can respond to values that a user supplies through a form.

The following helper function constructs an HTML form. The `label` argument is a string to show the user. The `next-url` argument is a destination for the form results. The `hidden` argument is a value to propagate through the form as a hidden field. When the user responds, the `"number"` field in the form holds the user's value:

```
(define (build-request-page label next-url hidden)
  `(html
    (head (title "Enter a Number to Add"))
    (body ([bgcolor "white"])
          (form ([action ,next-url] [method "get"])
                ,label
                (input ([type "text"] [name "number"]
                                      [value ""]))
                (input ([type "hidden"] [name "hidden"]
                                        [value ,hidden]))
                (input ([type "submit"] [name "enter"]
                                        [value "Enter"])))))))
```

Using this helper function, we can create a servlet that generates as many "hello"s as a user wants:

```
(define (many query)
  (build-request-page "Number of greetings:" "/reply" ""))

(define (reply query)
  (define n (string->number (cdr (assq 'number query))))
  `(html (body ,@(for/list ([i (in-range n)])
                   " hello"))))

(hash-set! dispatch-table "many" many)
(hash-set! dispatch-table "reply" reply)
```

Here's the whole program so far in plain text: step 6.

As usual, once you have added these to your program, update with `(enter! "serve.ss")`, and then visit `http://localhost:8081/many`. Provide a number, and you'll receive a new page with that many "hello"s.

10 Limiting Memory Use

With our latest `"many"` servlet, we seem to have a new problem: a malicious client could request so many "hello"s that the server runs out of memory. Actually, a malicious client could also supply an HTTP request whose first line is arbitrarily long.

The solution to this class of problems is to limit the memory use of a connection. Inside `accept-and-handle`, after the definition of `cust`, add the line

```
(custodian-limit-memory cust (* 50 1024 1024))
```

Here's the whole program so far in plain text: step 7.

We're assuming that 50MB should be plenty for any servlet. Due to the way that memory accounting is defined, `cust` might also be charged for the core server implementation and all of the libraries loaded on start-up, so the limit cannot be too small. Also, garbage-collector overhead means that the actual memory use of the system can be some small multiple of 50 MB. An important guarantee, however, is that different connections will not be charged for each other's memory use, so one misbehaving connection will not interfere with a different one.

So, with the new line above, and assuming that you have a couple of hundred megabytes available for the `mzscheme` process to use, you shouldn't be able to crash the web server by requesting a ridiculously large number of "hello"s.

Given the `"many"` example, it's a small step to a web server that accepts arbitrary Scheme code to execute on the server. In that case, there are many additional security issues besides limiting processor time and memory consumption. The `scheme/sandbox` library provides support to managing all those other issues.

11 Continuations

As a systems example, the problem of implementing a web server exposes many system and security issues where a programming language can help. The web-server example also leads to a classic, advanced Scheme topic: *continuations*. In fact, this facet of a web server needs *delimited continuations*, which PLT Scheme provides.

The problem solved by continuations is related to servlet sessions and user input, where a computation spans multiple client connections [Queinnec00]. Often, client-side computation (as in AJAX) is the right solution to the problem, but many problems are best solved with a mixture of techniques (e.g., to take advantage of the browser's "back" button).

As the multi-connection computation becomes more complex, propagating arguments through `query` becomes increasingly tedious. For example, we can implement a servlet that takes two numbers to add by using the hidden field in the form to remember the first number:

```
(define (sum query)
  (build-request-page "First number:" "/one" ""))

(define (one query)
  (build-request-page "Second number:"
                      "/two"
                      (cdr (assq 'number query))))

(define (two query)
  (let ([n (string->number (cdr (assq 'hidden query)))]
        [m (string->number (cdr (assq 'number query)))])
    `(html (body "The sum is " ,(number->string (+ m n))))))

(hash-set! dispatch-table "sum" sum)
(hash-set! dispatch-table "one" one)
(hash-set! dispatch-table "two" two)
```

Here's the whole program so far in plain text: step 8.

While the above works, we would much rather write such computations in a direct style:

```
(define (sum2 query)
  (define m (get-number "First number:"))
  (define n (get-number "Second number:"))
  `(html (body "The sum is " ,(number->string (+ m n)))))

(hash-set! dispatch-table "sum2" sum2)
```

The problem is that `get-number` needs to send an HTML response back for the current connection, and then it must obtain a response through a new connection. That is, somehow it needs to convert the page generated by `build-request-page` into a `query` result:

```
(define (get-number label)
  (define query
    ... (build-request-page label ...) ...)
  (number->string (cdr (assq 'number query))))
```

Continuations let us implement a `send/suspend` operation that performs exactly that operation. The `send/suspend` procedure generates a URL that represents the current connection's computation, capturing it as a continuation. It passes the generated URL to a procedure that creates the query page; this query page is used as the result of the current connection, and the surrounding computation (i.e., the continuation) is aborted. Finally, `send/suspend` arranges for a request to the generated URL (in a new connection) to restore the aborted computation.

Thus, `get-number` is implemented as follows:

```
(define (get-number label)
  (define query
    ; Generate a URL for the current computation:
    (send/suspend
      ; Receive the computation-as-URL here:
      (lambda (k-url)
        ; Generate the query-page result for this connection.
        ; Send the query result to the saved-computation URL:
        (build-request-page label k-url ""))))
  ; We arrive here later, in a new connection
  (string->number (cdr (assq 'number query))))
```

We still have to implement `send/suspend`. Plain Scheme's `call/cc` is not quite enough, so we import a library of control operators:

```
(require scheme/control)
```

Specifically, we need `prompt` and `abort` from `scheme/control`. We use `prompt` to mark the place where a servlet is started, so that we can abort a computation to that point. Change `handle` by wrapping an `prompt` around the call to `dispatch`:

```
(define (handle in out)
  ....
    (let ([xexpr (prompt (dispatch (list-ref req 1)))])
      ....))
```

Now, we can implement `send/suspend`. We use `call/cc` in the guise of `let/cc`, which captures the current computation up to an enclosing `prompt` and binds that computation to an identifier—`k`, in this case:

```
(define (send/suspend mk-page)
  (let/cc k
    ...))
```

Next, we generate a new dispatch tag, and we record the mapping from the tag to `k`:

```
(define (send/suspend mk-page)
  (let/cc k
    (define tag (format "k~a" (current-inexact-milliseconds)))
    (hash-set! dispatch-table tag k)
    ...))
```

Finally, we abort the current computation, supplying instead the page that is built by applying the given `mk-page` to a URL for the generated tag:

```
(define (send/suspend mk-page)
  (let/cc k
    (define tag (format "k~a" (current-inexact-milliseconds)))
    (hash-set! dispatch-table tag k)
    (abort (mk-page (string-append "/" tag)))))
```

When the user submits the form, the handler associated with the form's URL is the old computation, stored as a continuation in the dispatch table. Calling the continuation (like a function) restores the old computation, passing the `query` argument back to that computation.

Here's the final program in plain text: step 9.

In summary, the new pieces are: `(require scheme/control)`, adding `prompt` inside `handle`, the definitions of `send/suspend`, `get-number`, and `sum2`, and `(hash-set! dispatch-table "sum2" sum2)`. Once you have the server updated, visit `http://localhost:8081/sum2`.

12 Where to Go From Here

If you arrived here as part of an introduction to PLT Scheme, then your next stop is probably §“**Guide**: PLT Scheme”.

If the topics covered here are the kind that interest you, see also §10 “Concurrency” and §13 “Reflection and Security” in §“**Reference**: PLT Scheme”.

Some of this material is based on relatively recent research, and more information can be found in papers written by the authors of PLT Scheme, including papers on MrEd [Flatt99], memory accounting [Wick04], kill-safe abstractions [Flatt04], and delimited continuations [Flatt07].

The PLT Scheme distribution includes a production-quality web server that addresses all of the design points mentioned here and more [Krishnamurthi07]. See §“**Web Server**: PLT HTTP Server”.

Bibliography

[Flatt99] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen, "Programming L

[Flatt04] Matthew Flatt and Robert Bruce Findler, "Kill-Safe Synchronization Abstractions," Programming Lang

[Flatt07] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen, "Adding Delimited and Compo

[Krishnamurthi07] Shriram Krishnamurthi, Peter Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias

[Queinnec00] Christian Queinnec, "The Influence of Browsers on Evaluators or, Continuations to Program Web Serv

[Wick04] Adam Wick and Matthew Flatt, "Memory Accounting without Partitions," International Symposium or