Implementation and Use of the PLT Scheme Web Server *

Shriram Krishnamurthi, Peter Walton Hopkins† and Jay McCarthy
Brown University

Paul T. Graunke‡, Greg Pettyjohn and Matthias Felleisen
Northeastern University

Abstract. The PLT Scheme Web Server uses continuations to enable a natural, console-like program development style. We describe the implementation of the server and its use in the development of an application for managing conference paper reviews. In the process of developing this application, we encountered subtle forms of interaction not directly addressed by using continuations. We discuss these subtleties and offer solutions that have been successfully deployed in our application. Finally, we present some details on the server's performance, which is comparable to that of the widely-used Apache Web server.

1. The Structure of Web Programs

Consider the following sequence of Web interactions:

1. A user examines a list of books.
2. The user requests the page for book A.
3. The user presses the "buy now" button on the page showing book A.

As expected, the application registers a purchase for book A. This behavior, which corresponds to a traditional console application, seems entirely reasonable. Consider, however, the following sequence with the same implementation (the W_i name browser windows):

1. The user examines a list of books (W_1).
2. The user requests the page for book A in a new browser window (W_2).
3. The user switches to W_1.
4. The user requests the page for book B in a new browser window (W_3).
5. The user switches to W_2 and presses the "buy now" button on it.

In some implementations, this sequence of interactions can result in a purchase of book B rather than of A. This is obviously not at all what the user intended.

* This research is partially supported by multiple NSF grants. Preliminary versions of parts of this material have appeared in print [11, 18, 20, 23].

† Current Affiliation: Google, Inc.

‡ Current Affiliation: Galois Connections, Inc.

This behavior is sadly not hypothetical. Several of the authors have encountered it on multiple Web sites, including major commercial applications such as travel sites [24]. This has resulted, for instance, in colleagues being booked on the wrong flights. One particular travel Web has since fixed their software, but only slightly: as of September, 2004, this scenario results in an error instead of the wrong purchase.

Why do Web applications, even those built by corporations that depend heavily on their Web presence, behave thus? Two factors complicate the development of Web software relative to traditional console software. The first is that the Web browser gives users the ability to traverse a web of interaction through the use of back and forward buttons, window cloning operations, and so on. The second is that, for scalability, the Web is built atop a stateless interface, which complicates the structure of interactive Web software.

How does the statelessness of the Web's CGI protocol [32] (and of related APIs such as that for Java's servlets [42]) influence program structure? A Web server invokes an application on request from a client, and sends the application's intermediate output to the client awaiting further inputs and other decisions. Because of statelessness, however, the application then *terminates*. It therefore becomes the responsibility of the application writer to store enough information—both the control context and data bindings—to then *restore* the computation if the user ever chooses to interact.[1] In response to this constraint, Web programmers have developed some standard patterns for codifying an interactive Web computation:

- Programmers develop programs that have several distinct, root-level entry points, one per interaction point. That is, the Web application either consists of several small programs or of one program that uses some piece of the input to decide which main function to call.

- Programmers use a variety of mechanisms, such as "hidden fields" and "cookies", to record the data and control contexts. These data must be conveyed from one fragment of computation to the next, either by storage on the server or by being sent to and returned from the client. Developers must be careful to not confuse these different mechanisms for each other.

While programmers try to adhere to these conventions, it is clear that following these rules produces unnatural—indeed, inverted—code. Because of this structural inversion, it becomes onerous to maintain and evolve code through bug fixes and requirements changes.

A semantic explanation of the first pattern is due to Christian Queinnec [38]. According to Queinnec, Web browsing in the presence of dynamic Web content generation is the process of capturing and resuming continuations.[2] With this suggestion in mind, we can now name the CGI program design pattern: continuation-passing style (CPS) [13]. Roughly

[1] In this paper, we use "interaction" to refer to one communication between a Web application and its user; "interaction point" to refer to a syntactic expression in the Web application that enables interaction to take place; and "computation" to refer to a run of the Web application, which usually consists of a sequence of interactions.

[2] This idea also appears in Hughes's paper on arrows [21] and Paul Graham's work on Viaweb [17].

speaking, a function in CPS receives its inputs and its continuation (another function); then it performs its task; and finally, it hands off the result of this task to the continuation. If each continuation has a name, turning a program in CPS into a CGI program is straightforward. When a function has performed its task it outputs the result and the name of the continuation. The browser, in response, submits this name and inputs to the server, and the program's entry point dispatches to the function of this name.

The dual problems of code inversion and improper data serialization cause many of the Web application failures we have discussed. We have therefore explored an alternative framework for writing Web applications and report on our results in this paper. Specifically, we present the PLT Scheme Web Server [18], a server written in PLT Scheme [12] that uses first-class continuations to enable direct-style programming [10] for Web applications. In addition to describing the server, we present its use as an application platform through several illustrative examples, including a production-quality application for conference paper management called CONTINUE. We also demonstrate that building this application has helped us design and integrate better paradigms for programming interactive Web applications.

In describing this work, the paper presents several contributions. We demonstrate weaknesses in the simple continuation-based pattern, and show how to extend it to richer patterns through new primitives. We show that the continuation-based framework naturally results in interesting functionality such as one-shot URLs. We discuss the nature of Web data and explain errors such as those discussed at the beginning of this section. Finally, we present a non-trivial system with two tiers—the server and CONTINUE—implemented entirely in PLT Scheme, as an illustration of a real-world use of a functional language. (Indeed, using the continuation-based programming style increases the functional nature of the application code.)

This paper is organized as follows. Section 2 describes the PLT Scheme Web Server. It describes static and dynamic document service followed by the rich set of control primitives offered by the server. It presents our resource management strategies, and finally offers a performance comparison to some other Web programming frameworks. Section 3 describes the CONTINUE application, highlighting an instance where the Web Server's primitives were especially or insufficiently useful. Section 4 identifies and then addresses the problem we encountered implementing CONTINUE: the use of one continuation to represent multiple futures depending on the user's choice. Section 5 summarizes what we have learned from several years of experience about programming style and its interaction with the different flows of data on the Web. The remaining sections discuss related work and offer concluding remarks.

2. The PLT Scheme Web Server

This section outlines the use of the PLT Scheme Web Server. We focus primarily on the generation of dynamic content, describing the primitives offered by the server. We also

```
(unit/sig () (import servlet^)
  (define TITLE "My first Web page")
  `(html (head (title ,TITLE))
         (body
          (h1 ,TITLE)
          (p "Hello, world!"))))
```

Figure 1. A servlet that defines a title and uses it to produce a simple Web page containing a message.

discuss some of the problems and solutions pertaining to resource management and browser-supported user interactions.

2.1. Serving Static Pages

A Web server responds to HTTP requests by reading Web pages from files on disk. The PLT Scheme Web Server uses the common approach of specifying a directory to be the root of static content and mapping URLs directly to the filesystem structure beneath. Section 2.7 discusses the performance of the server on static content.

2.2. Serving Dynamic Content

The server specifies a separate directory for dynamic content generators ("servlets"). Each file in the directory is a Scheme program that defines a PLT Scheme module called a *unit* [15]. When invoked, this unit produces a Web page.

Servlets can leverage Scheme's list-manipulation primitives by describing Web pages as X-expressions [11, 25]—S-expressions that represent XML [6]—instead of as strings.[3] Figure 1 is a complete servlet that uses **quasiquote** [37] (abbreviated by a single open quote or "back-quote") to create an XHTML[4] page, with **unquote** (abbreviated by a comma) to reference the definition of *TITLE*. (The suffix '^' in *servlet^* is a syntactic convention used to designate names of interfaces.) More general servlet forms, not shown here, allows the use of other content generation mechanisms, ranging from more sophisticated HTML-creation tools to servlets that produce content in formats entirely unrelated to HTML.

Units are first-class values, so servlets can be stored in a cache by the server. This cache significantly speeds up execution by avoiding I/O, parsing, and compilation for each invocation. In addition, it also allows servlets to maintain local state across invocations. A servlet that takes advantage of this is shown in figure 2.

[3] There are other Scheme representations of XML including those by Kiselyov [22] and Nørmark [33].

[4] XHTML [49] is a reformulation of HTML 4.0 into an XML language. We employ XHTML because it forces the use of balanced tags, which simplifies recursive document generation by eschewing special cases.

```
(define count 0)
(unit/sig () (import servlet^)
  (set! count (add1 count))
  `(html (head (title "Counter"))
         (body
          (p "This page was generated by a computer program.")
          (p "The current count is " ,(number→string count)))))
```

Figure 2. A servlet to demonstrate state saved across invocations.

send/suspend : (URL → X-Expression) → HTTP-Request
request-bindings : HTTP-Request → listof Binding
exists-binding : Symbol × listof Binding → Boolean
extract-bindings : Symbol × listof Binding → listof String
extract-binding/single : Symbol × listof Binding → String

Figure 3. Contracts for the PLT Scheme Web Server functions.

2.3. Servlets that Interact

As we noted above, interactions between users[5] and servlets can be understood as capturing and resuming continuations. Our server takes advantage of Scheme's first-class continuations to provide servlet developers with a key primitive for creating interaction points: *send/suspend*.

A servlet can use *send/suspend* to send an XHTML page to the client and to suspend execution until the client issues a request.[6] The *send/suspend* function captures and stores the servlet's continuation, suspends the servlet, then sends the XHTML page (which is the heart of the response) to the client's browser. The function also generates a unique URL that it associates with this continuation. The servlet typically embeds this URL in the generated Web page; when the user responds to the page by clicking a link to the unique URL or submitting a form whose `action` attribute is the unique URL, the server resumes the continuation and *send/suspend* returns with the user's request. As we demonstrate in section 3.1, however, there are cases when a servlet may choose to not use the URL in the generated document, but rather send it to the user by other means.

As figure 3 shows, *send/suspend* consumes a function of one argument, which itself consumes a unique URL and generates an XHTML page. The unique URL, conventionally

[5] Two categories of people "use" a server: the end-users who interact with a Web application running on the server and the developers who create them. In this paper, we will use the term "users" to refer to the former group, and "developers" for the latter.

[6] In the terminology of HTTP, a message from the browser to the server is a *request*, while one from the server to the browser is a *response*. This does not always match the colloquial use of these terms, because the user operating the browser may be "responding" to information from the server, e.g., choosing a book from a list of presented choices. Nevertheless, in this paper we adhere to the HTTP convention.

```
(unit/sig () (import servlet^)
  ;; helper function
  (define (get-number which-number)
    (define req
      (send/suspend
       (lambda (k-url)
         `(html (head (title ,which-number " number"))
                (body
                 (form ([method "post"] [action ,k-url])
                       "Enter the " ,which-number " number: "
                       (input ([type "text"] [name "number"]))
                       (input ([type "submit"])))))))))
    (string→number (extract-binding/single 'number (request-bindings req))))
  ;; main body
  `(html (head (title "Product"))
         (body
          (p "The product is: "
             ,(number→string (* (get-number "first") (get-number "second")))))))
```

Figure 4. An interactive servlet for curried multiplication.

called *k-url*, serves as the key to the stored continuation: when the browser requests it, the continuation is resumed and *send/suspend* returns. Section 2.5 will describe how the URLs are constructed and how the server maps requests for these URLs to their associated continuations.

The return value from *send/suspend* holds information from the browser's request, including the requested URI, HTTP headers, and a list of bindings. The bindings, accessible via the *request-bindings* function, are the combined parsed representations of the URL's query string and HTTP POST data, which are the two mechanisms for transmitting form data to the server. The value or values bound to a particular key can be retrieved with the *extract-bindings* and *extract-binding/single* functions; the latter function is more convenient when the key is guaranteed to be present and unique (and checks for this fact).

Figure 4 presents Queinnec's illustrative servlet. This code implements curried multiplication, asking for each number on a separate page. Notice how this Web program is written in a natural, direct style, despite its two interaction points. In general, direct-style programs naturally accommodate the flow of information between client and server. These programs are easier to match to specifications, to validate, and to maintain than their equivalent CPS or CGI versions.

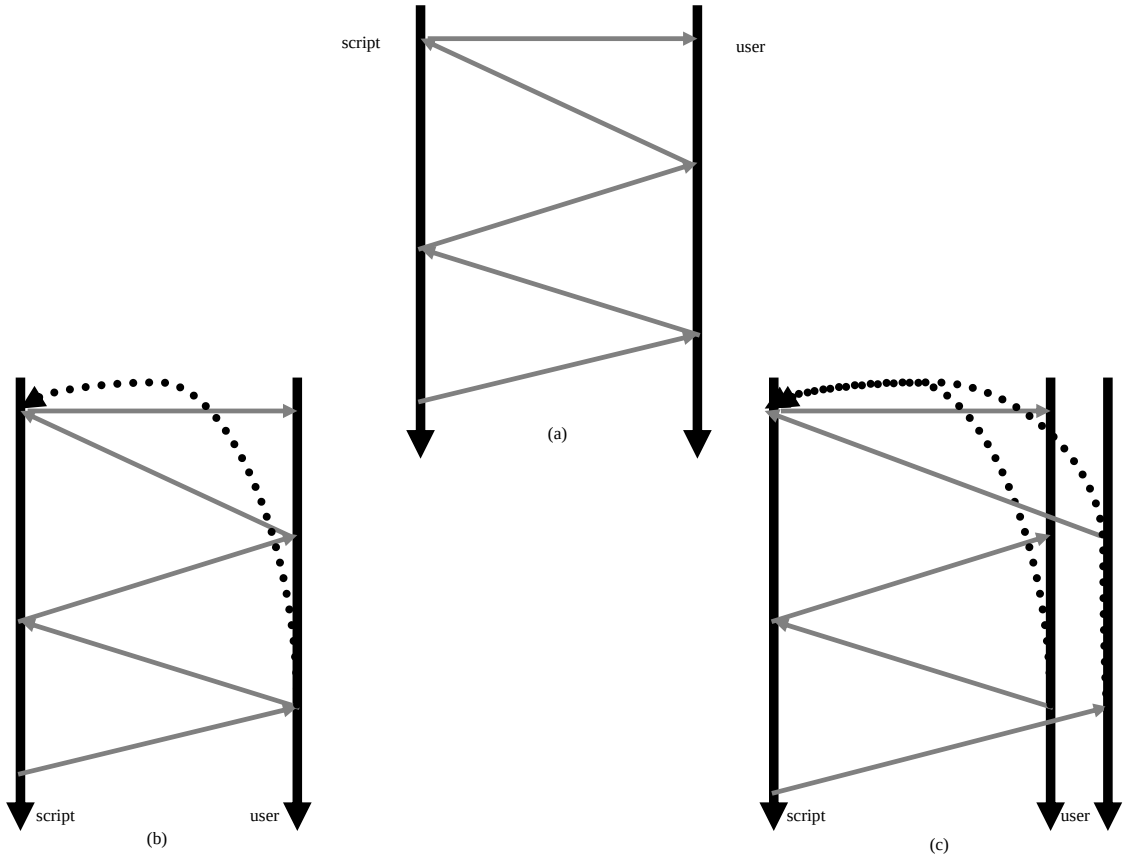


Figure 5. The effect of user interactions.

2.4. Additional Interaction Primitives

Besides allowing sequential execution, Web browsers give users the flexibility to bookmark intermediate continuations and explore various choices. Figure 5 presents this graphically. Here, each vertical bar represents the *relative* timeline of an agent (i.e., the sequence of events), with time increasing downward. The horizontal bars represent a message sent from the tail to the head of the arrow. The idea is that each message *resumes* the other computation in the state it was in earlier—as if it had suspended in time waiting for this message. Figure 5 (a) presents the traditional client-server sequence of interactions, which reflects classical coroutine behavior. Figure 5 (b) shows the use of the back-button (the dotted-line). This allows the user to return to a prior coroutine resumption point. Pressing the back-button is, furthermore, a *silent action*, because the browser does not notify the server of the button-press; the application must either infer this from the next user interaction, or be robust enough to not care. Figure 5 (c) illustrates the use of cloning,

Table I. Additional interaction primitives.

primitive	contract	resumable	backtrackable
send/suspend	(URL → X-Expression) → HTTP-Request	yes	yes
send/forward	(URL → X-Expression) → HTTP-Request	yes	no
send/back	X-Expression → Void	no	yes
send/finish	X-Expression → Void	no	no

which now generates two copies of the same running computation, each with additional browsing power.

This navigation power demands a careful specification of program behavior. Although it is desirable to allow a user to use the back button and to resubmit forms while a computation is in progress, once the computation is complete, resubmission of intermediate forms may cause problems. When a program finishes interacting with the user it may record the final outcome of the computation by updating persistent state, such as a database. These updates must happen at most once to prevent catastrophes. For instance, one of the authors of this paper encountered this type of error when renewing two Internet domain name registrations. The penultimate page of the registration program indicated that the user should wait for the server to finish processing the renewal request. After a moment, it automatically proceeded to the final page, confirmed the renewal, and billed the author's credit card. Hitting the back button (to renew the second domain) returned to the processing page, which billed the credit card again, renewing the first domain for an additional year. Another author had a similar experience when saving a receipt page: the act of saving reloaded the page, which resulted in a double-billing.

To support fine-grained control over user interactions, the PLT Scheme Web Server provides the following primitives in addition to *send/suspend*. To support terminating a computation, the server provides *send/finish*. When a program invokes this primitive, the server releases all continuations associated with that servlet instance. When a user attempts (e.g., via a bookmark) to access a reclaimed continuation, a page directs them to restart the computation. If a servlet must prevent a consumer from backtracking amidst a session—as the domain registar should have done after updating the persistent state—the *send/forward* primitive will clear the servlet's suspended continuations before inserting a continuation to correspond to the current interaction. This enables consumers to proceed forward, but not repeat previous interactions. The most basic primitive is *send/back*, which does not generate a fresh URL; the other primitives are built atop *send/back* (see, for instance, figure 15). Table I compares the navigational capabilities of these interaction primitives.

These operators do not properly cover one other standard browser operation, reloading, which is triggered not only by the reload button but also by actions such as saving, printing, cloning, or even hitting navigation buttons like back and forward (if the page has not been cached or the cached copy has expired). Reloading re-sends the request that generated the current page, thereby re-executing code starting from the previous interaction

point. Because of state, however, this fragment of computation may not produce the same output as before; furthermore, it may have side-effects (which can be arbitrarily complex, such as charging a credit-card). This is a well-known problem with the HTTP protocol, to which a popular solution is to use the "post-redirect-get" pattern. Herman [19] shows that this pattern can be easily encoded as an abstraction atop the PLT Scheme Web Server. In general, the later a servlet can defer updates to external persistent state (such as a database), the more they can avoid unexpected interactions in the face of user behavior. Where such updates happen, it is best to use a primitive such as *send/forward* to invalidate the prior interactions.

2.5. Implementation

The Web server is written entirely in PLT Scheme. The core of a Web server is a wait-serve loop that waits for requests on a designated TCP port. For each request, it creates a thread that serves the request. Then the server recurs:

```
;; server-loop : TCP-Listener → Void
(define (server-loop listener)
  (define-values (ip op) (tcp-accept listener))
  (thread (lambda () (serve-connection ip op)))
  (server-loop listener))
```

For each request, the server parses the first line and the optional headers:

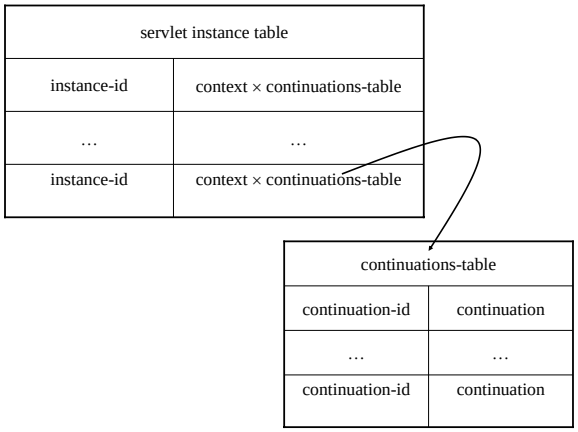
```
;; serve-connection : Input-port × Output-port → Void
(define (serve-connection ip op)
  (define-values (meth url-string major-version minor-version)
    (read-request ip op))
  (define headers (read-headers ip op))
  (define url (string→url url-string))
  (define host (find-host (url-host url) headers))
  (dispatch meth host port url headers ip op))
```

The procedure *read-request* consumes a request from an input port, parses it, and responds with the request type (GET or PUT), the URL, and protocol versions. If parsing fails, it raises an exception and closes both ports. The actual code also includes a loop to support HTTP stay-alive, but the code above represents its essence.

When the request is for a static file, a dispatcher uses this information to find the correct file corresponding to the given URL. If it can find and open the file, the dispatcher writes the file's contents to the output port; otherwise it writes an error message.

The implementation of *send/suspend* captures the current continuation, stores it in a table, and generates a URL that contains a reference to that table entry. The table that stores continuations actually has two tiers. The first tier tracks instances of servlets: every initiation of a servlet creates a new entry in this tier. (This corresponds to the notion of a "session" used by some Web APIs, but does not require the user to explicitly instantiate the

session.) The second tier tracks continuations within an instantiation of a single servlet. In essence, the tables are structured as follows:



The *context* keeps track of the current ports and the content of the HTTP-Request. When a user submits a Web form, the server invokes the corresponding continuation with the context information. A subsequent use of *send/suspend* within that execution extends the same continuation table.

A typical continuation URL looks like this:

```
http://www/servlets/send-test.ss;id38*1*839800468
```

This URL has two principal pieces of information for resuming computation: the servlet instance-id (`id38`) and the continuation-id (`1`). The random number at the end serves as a nonce to prevent users from guessing continuation URLs.

2.6. Resource Management

The CGI interface is designed to make the Web server (the application, not the machine that runs it) stateless: because the servlet must terminate upon every interaction, it must store all its pertinent state either in permanent storage on the server machine or transmit these data to the client. In either case, the Web server itself is not responsible for the state, which means if the user never resumes a computation, the server does not bear the burden of reaping resources associated with that user. In practice, however, this simply shifts the burden to the servlet and its creator.[7]

By storing continuations on the server, we introduce a garbage collection problem. Every URL created by *send/suspend* is a reference to a value in the server's heap. Without further restrictions, garbage collection cannot be based on reachability (because the server cannot,

[7] Naturally, data stored on the heap are susceptible to server failures. PLT Scheme provides interfaces to persistent storage on the server (such as ODBC-compatible databases) and client (such as cookies [26]) so servlets can better tolerate faults.

for instance, read the bookmark file of every past user). To make matters worse, these continuations could also hold on to resources, such as open files or TCP connections (to a database, for example), which the server may need for other programs.

The PLT Scheme Web Server presents two mechanisms that servlet authors can employ to manage resources:

- Each instance of a servlet has a predetermined lifetime after which its continuations are automatically reaped. Each use of a continuation updates this lifetime. A running computation may change the duration of this lifetime, which is measured in real time, by invoking the *adjust-timeout!* operator; as a special case, this timeout can be set to infinity.

- When the servlet terminates, all the continuations associated with that instance of the servlet are released.[8] A user who attempts to access a reclaimed continuation is redirected to an error page that allows the user to restart the computation. This policy allows the server to manage resources and to prevent the user from restarting computations that have completed.

The two-tier structure of the tables facilitates clean-up. When the time limit for an instance of a servlet expires or when the servlet computation terminates, the instance is removed from the servlet instance table. This step, in turn, makes the corresponding sub-table of continuations inaccessible and thus available for garbage collection.

In principle, a mere timeout is not sufficient. While the timeout has the effect of removing references to the continuations, this does not cover all the possible resources the servlet instance may be consuming: for instance, it does not cover ports that refer to an external service such as a database. As a result, these resources would still appear to be consumed, making them unavailable to other servlets. To avoid this leakage, the Web server needs to treat the servlet *instance* as a resource principal, thereby accounting for and eventually releasing all the resources it consumes.

Fortunately, PLT Scheme has facilities for fine-grained resource management: *custodians* [16]. A custodian records a collection of resources such as files and network ports, sockets, and network listeners, and provides a single primitive for closing them and returning them to the system. The timeout mechanism for shutting down the servlet works because the reaper and the servlet's thread share the same custodian.

There are actually numerous custodians in use at any instant by the server. In addition to the custodians governing each servlet instance, there are custodians for each connection and for the server itself. Figure 6 depicts the layout of custodians within the Web server. The entire server runs within a single, top-level custodian, so the administrator can easily shutdown the entire server. Each time the server listens for a new TCP connection on the HTTP port, the server first creates a new custodian to manage the connection. After creating new IO ports for the connection, the server loop starts a new thread within the connection custodian to process the connection. That is, even though in our particular

[8] The server uses locks to guard accesses to its data structures. It does not, however, automatically impose a locking discipline on the servlets.

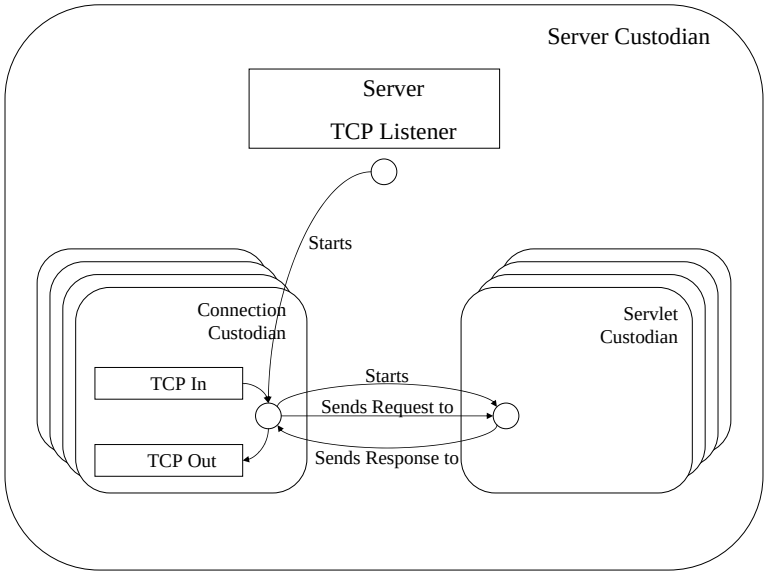


Figure 6. Organization of custodians.

custodian strategy there is a one-to-one correspondence between threads and custodians, the parent thread allocates new network ports within the child's custodian before the child thread exists. Thus, the separation of threads and custodians helps with the situation. This illustrates why custodians, or resource management in general, should not be identified with individual threads.

An additional reason custodians should not correspond to threads is related to the resource utilization of servlet instances. Because servlet instances must handle multiple requests, and thus multiple connections, the birth and death of connections and servlets proceed independently.

Servlet instances are like procedures that transform requests into responses. As mentioned earlier, in version 1.1 of HTTP there may be multiple requests per connection. Similarly there will be multiple responses per connection in these cases. During the course of operation, there may be connection errors, or chunked encoding demands, and, of course, the servlet may not be programmed correctly and may thus leave the connection in an erroneous state.

To deal with these complications and their resource utilization demands, the thread for each connection creates a new custodian for each new servlet. Then, the connection thread locates the appropriate continuation and activates it. The value that the connection thread receives from this invocation is the response that is sent to the user. The connection thread may invoke the servlet multiple times, perform operations on either the request or the response, and manage other aspects of the connection that is unrelated to the computation that the servlet performs, protecting the servlet from such concerns.

2.7. Performance

It is easy to write compact implementations of systems with high-level constructs, but we must demonstrate that we don't sacrifice performance for abstraction. In this section, we therefore present experimental evidence in support of our server's performance. Our initial focus has been on small- to medium-sized groups of users, such as a small online communities, research groups, and small intranets.[9] All experiments were conducted on Linux 2.6.7 using Apache/1.3.33, libapache-mod-perl 1.29.0.3-8, and PLT Scheme version 299.402. We used all these systems in their standard configurations.

2.7.1. *Static Content*

In principle, we would like our server to transfer content from files at about the same rate as a widely-used server such as Apache [45]. We believed that this goal was within reach because most of the computational work involves parsing HTTP requests, reading data from disk, and copying bytes to a (network) port.[10]

To verify this conjecture, we compared our server's performance to that of Apache on files of three different sizes. For each test, the client requested a single file repeatedly. This minimized the impact of disk speed because the underlying buffer cache should keep small files in memory. Requests for different files would even out the performance numbers according to Amdahl's law [1] because the total response time would include an extra disk access component that would be similar for both servers.

The results in table II show how the PLT Scheme Web Server compares against Apache. The client and server software each ran on an AMD Athlon 64 3000+ processor with 512 Mbytes of memory, running Linux 2.4.27, connected by a standard 100 Mbit/s Ethernet connection. Each ratio column compares the previous two (PLT/Apache). The results were obtained using the S-client measuring technology [3]. For the historically most common case [2]—files between six and thirteen kB—our server's performance is comparable to Apache's, especially for small numbers of users. For larger files, which are now more common due to increasing use of multimedia, the performance of the two servers is quite comparable. In particular, the files statically served by our Continue Web application—academic papers in PDF format—range from several hundred kilobytes to more than a megabyte, and the number of clients tends to be small (roughly, the number of program committee members assigned to review each paper).

For one and ten kB files, the measurement client's requests caused both servers to consume all available CPU cycles. For the larger 100 kB files, both servers drove the network card at full capacity, so additional measurements would not be instructive.

The space usage of both systems was comparable. The PLT Scheme Web Server consumes about 13 megabytes of memory after thousands of static requests. Apache has a comparable fixed cost of about 25 megabyes, and an additional cost of about 3 megabytes per child process. The difference between these numbers does not appear to be significant.

[9] As of September 2005, a company in the UK is considering offering Web hosting for users who want to write their applications atop the PLT Scheme Web Server.

[10] This assumes that the server does not use a large in-memory cache for frequently-accessed documents.

2.7.2. *Dynamic Content*
A Web server provides operating system-style services. Like an operating system, a server runs programs (namely servlets). Like an operating system, a server should protect these programs from each other. And, like an operating system, a server manages resources (such as network connections) for the programs it runs. Most existing Web servers rely on the underlying operating system to implement these services: they create a fresh OS process for each incoming request which necessitates the creation of a new address space, loading code, and so on. The PLT Scheme Web Server instead uses user-level threads,[11] and relies on the operating system-like facilities provided by the underlying language to offer protection to individual servlet instances (as discussed in section 2.6). For these two reasons, we should expect higher performance from our Web server than from conventional servers that use CGI.

To evaluate dynamic performance, we measured our server against CGI programs written in C and in Perl, run by the `mod_perl` extension to Apache. Table III clocks a C program's binary in CGI and a Perl script in `mod_perl` against a Scheme script producing the same data. (The table does not contain performance figures for responses of 100 kB and larger because for those size the network bandwidth becomes the dominant factor just as with static files.) The C version measures the maximum performance developers can expect of CGI programs in Apache, while the `mod_perl` version provides a more realistic benchmark of how Apache is used to serve CGI programs for the same class of applications as PLT.

Our experiments confirm that our server handles more connections per second than CGI programs written with `mod_perl` in the limit, and particularly for small response sizes.

The table also indicates that each server's performance degrades with larger responses. We conjecture that this owes to each server copying the response multiple times. We note that this problem is orthogonal to the slowdown our server experiences in the face of concurrent clients, but we need to perform additional experiments to study this question in more depth. Of course, as computations become more intensive, the comparison becomes one of compilers and interpreters rather than of servers and their protocols.

These dynamic experiments were designed to test throughput. Because these tests do not engage in meaningful interactions, the space consumption numbers are not relevant. We instead report on space consumption in the application context below.

In future we expect to rework our IO strategy to handle higher request rates, potentially in a style similar to Flash [35].

2.7.3. *Application Resource Usage*
In addition to benchmarks, it is useful to analyze the server on a working application. Here we present statistics gathered from uses of CONTINUE.

When running CONTINUE we disable timeouts to maximize usability and minimize frustration, especially close to the deadline. A conference in 2003 of 60 paper submissions consumed 49 CPU minutes and 159 megabytes. For a more recent conference (early 2005) of

[11] The term "user" is used here in the conventional sense when describing thread packages, i.e., as opposed to system-level threads.

Table II. Performance for static content service (requests/second).

	1kB file			10kB file			100kB file		
Clients	PLT	Apache	Ratio	PLT	Apache	Ratio	PLT	Apache	Ratio
2	368.3	311.9	118.1%	368.9	299.4	123.2%	298.0	174.0	171.2%
4	518.9	633.0	82.0%	320.0	402.2	79.6%	319.8	243.3	131.4%
8	325.0	679.9	47.8%	311.2	673.2	46.2%	281.4	300.9	93.5%
16	310.0	740.0	41.9%	315.8	724.2	43.6%	273.2	379.9	71.9%
32	329.2	1099.0	30.0%	311.0	920.0	33.8%	287.3	430.2	66.7%
64	333.4	1357.7	24.6%	314.8	914.1	34.4%	272.5	467.5	58.2%
128	349.8	1187.5	29.4%	254.9	912.5	27.9%	301.5	481.2	62.6%

Table III. Performance for dynamic content generation (requests/second).

	C CGIl		ModPerl		PLT	
Clients	1kB	10kB	1kB	10kB	1kB	10kB
8	33.06	30.38	32.05	29.18	146.50	23.42
16	33.29	30.58	33.09	20.80	153.80	22.80
32	33.55	30.74	26.90	11.90	155.80	23.24

40 submissions, the submission and review process required 4427 hits, each of which would have created a fresh, unreaped continuation.[12] For this conference, the server consumed 72 minutes of processor time and 63.8 megabytes of memory. Currently (late 2005), CONTINUE is concurrently serving four conferences with a total of 256 papers. These have generated just over 25,000 requests. This has consumed 280 minutes of CPU time and 96 megabytes of memory. As such, therefore, the memory consumed by this application is extremely reasonable by modern server standards, and has further been reducing over time reflecting improvements to the server and to the underlying implementation.

3. CONTINUE

The CONTINUE application, written by the first three authors, supports the paper submission and review phases for academic conferences. This application, whose user interface is

[12] The ratio of hits to papers might look abnormally high, but recall that continuations are generated not just for each paper, but for each view: every login, listing of unreviewed papers, and so on generates a fresh continuation.

```
;; ask-email : () → Email
;; ask-paper-info : Email → Paper-Info
;; save-paper : Email × Paper-Info → Void
;; send-confirmation-email : Email × Paper-Info → Void
;; show-confirmation-page : Paper-Info → Void

(define email (ask-email))
(define paper-info (ask-paper-info email))
(save-paper email paper-info)
(send-confirmation-email email paper-info)
(show-confirmation-page paper-info)
```

Figure 7. The skeleton of CONTINUE's paper-submission code.

discussed elsewhere [23], has been used successfully for several conferences and workshops spanning diverse communities, including the Computer Security Foundations Workshop, the International Symposium on Software Testing and Analysis, the ACM Symposium on Software Visualization, and others. Its performance and stability demonstrate the utility of the PLT Scheme Web Server.

Most of the application is written in a functional style. The use of *send/suspend* makes it possible to create natural abstractions over Web interactions. In this section, we present two families of code examples from CONTINUE. The first presents a strength of the continuation-based paradigm for implementing useful functionality. The second points to a deficiency in the straightforward use of continuations; section 4 presents the solution eventually deployed in CONTINUE.

3.1. Scenario: Implementing Paper Submissions

The paper submission process is a clear example of the programming convenience that continuation-based Web programming engenders. The top-level code for paper submission is shown in figure 7. Thanks to *send/suspend*, we can write this code in a direct, mostly-functional style. Both *ask-verified-email* and *ask-paper-info* interact with the user, but that does not prevent them from being treated as abstractions that return values. In contrast, in a traditional CGI or Java servlet Web application, a function would not be able to both send a page to the user and handle the user's subsequent request.

An author who visits the paper submission servlet is first prompted to enter a name and email address in a form. Submitting this form leads to a second form, for entering metadata and actually uploading the paper. This form is generated and processed by the *ask-paper-info* function. Once the paper is saved to persistent storage with *save-paper*, the author sees a confirmation page and receives a confirmation email.

The *ask-email* function is shown in figure 8. It first presents the paper author with a form for entering an email address. If the author submits the form without entering an

```
(define (ask-email)
  (define contract-request
    (send/suspend
     (lambda (k-url)
       `(html ...
              (form ([action ,k-url])
                    ...
                    (input ([name "email"]))
                    ...)
              ...))))
  (define contact-bindings (request-bindings contact-request))
  (define email (extract-binding/single 'email contact-bindings))
  (if (string=? email "")
      (ask-email)
      email))
```

Figure 8. A simple implementation of the *ask-email* function.

email address, it asks again. Otherwise, it returns the email address to its caller, which in this case is the code from figure 7.

While this implementation of *ask-email* is acceptable in theory, it is unsatisfactory in practice because it makes no attempt to validate the email address. The program chair would ideally like to have a verified email address for every submitted paper, so we implement the following strategy. When the author submits an email address, the servlet sends a message to that address. The response Web page simply requires the author to read email; it does not provide any means to continue the dialog. The author can, however, use the back button to re-enter the address, if the original submission contained an error.

The email message contains a URL that the author must submit via a browser. Since this URL encodes the continuation, the act of submitting it takes the place of clicking on a Web page element, and thereby resumes the dialog. Because the PLT Scheme Web Server generates unique URLs, the servlet can reasonably assume that

- the email message was legitimately read, and
- the recipient of the email message was indeed the person who intended to submit a paper to the conference.

In short, the act of resuming the computation itself validates the author's email address and intent.[13]

Figure 9 presents the corresponding code, which has two noteworthy aspects:

[13] This pattern of authentication is used much more broadly than just for conference software. It commonly occurs when registering users to mailing lists and other services.

```
(define (ask-email) ;; now returns a validated address!
  (define contact-request
    (send/suspend
     (lambda (k-url)
       `(html ...
              (form ([action ,k-url])
                    ...
                    (input ([name "email"]))
                    ...)
              ...))))
  (define contact-bindings (request-bindings contact-request))
  (define email (extract-binding/single 'email contact-bindings))
  (if (string=? email "")
      (ask-email)
      (begin
        (send/suspend ; return value is irrelevant
         (lambda (mailed-url)
           (send-verifying-email email mailed-url)
           `(html ...)))
        email)))
```

Figure 9. A more robust implementation of *ask-email* that verifies the author's email address.

– Continuations preserve lexical environments, so no extra effort is needed on the part of the developer to preserve the value of *email*, even though it is used on either side of an interaction point.

– The URL that *send/suspend* supplies to the page-generating function (called *mailed-url* in this code) need not be used in the resulting Web page.[14] Though the Web page generated by the second use of *send/suspend* in *ask-email* is a dead end, the continuation of this Web interaction point is available via the URL sent to the author's email address.

The continuation of the *send/suspend* in the main body of the procedure simply returns the email address for use in the rest of the computation.

3.2. Scenario: Multiple Links on a Single Page

The paper submission process has a linear flow (save for barring the submission of empty email addresses), so it can be implemented in a straightforward manner with *send/suspend*.

[14] We expose the *send/suspend* primitive for such uses, even though in the common case it is easier to employ an abstraction that automatically binds the continuation, endows the generated form with an `action` attribute, and binds the attribute to the continuation.

This is not, however, the case for all servlets. In this section, we examine two examples from CONTINUE that highlight the problem caused by more complex control flows.

3.2.1. *Generating Links to Papers*
Consider the code that generates the reviewer interface to CONTINUE (see figure 10). In this interface, each paper title is a link to a page that shows that paper's reviews. Since the number and content of reviews changes over time, clicking each link causes further dynamic content generation. In principle, therefore, each link should have a unique URL that represents the generation of that paper's current review information.

In keeping with the notion that there is always a single continuation, the PLT Scheme Web Server generates only one URL at each suspension point. Since each paper's link is derived from the same URL, however, each link will resume the same continuation. The continuation must therefore contain code to dispatch between these choices; the dispatcher must interpret, from the browser's request, which link the user clicked on.[15]

Early versions of CONTINUE generated the links by appending a query string to the URL generated by the server. The dispatch code used the request bindings to determine which paper the user clicked on, then called the review page function with the paper number as an argument. Figure 11 shows the relevant pieces of this implementation. (The *format* command replaces instances of the substring `"~a"` in its first argument, the format string, with the content of the additional arguments provided to it.)

3.2.2. *Implementing Tabs*
The dispatching process in figure 11 is relatively simple, because the papers are automatically indexed by paper numbers. Sometimes, however, the developer must generate an artificial index for the purpose of dispatching. Consider the navigational tabs included on pages generated by CONTINUE, such as those shown in figure 10. In the interest of good program design we wish to abstract over their presentation, which means we intend to reuse both the code that generates them and the code that dispatches on them.

The code in figure 12 augments the code in figure 11 to implement the tabs shown in figure 10. The addition of tabs demonstrates that dispatching is not always immediate. When an indexing scheme is not readily available, the developer has to create more intricate dependencies between the code generating the links and the code dispatching on the results. In effect, the developer is manually programming a multiplexer (augmenting the generated URLs) and a demultiplexer (the dispatcher). The multiplexer and demultiplexer must conspire to satisfy a common invariant, so one must not be changed without updating the other to match. This is shown pictorially in figure 13. Writing this dispatcher therefore requires the manual application of a programming pattern. Furthermore, the developer should, in principle, validate the index on submission of the user's request: a malicious user might have modified the indices, which may cause the software to expose protected information or even to crash.

[15] The dispatch step is an example of "leakage," a term coined by Lawall and Friedman [28] for particular instances of a function caller repeating the work of the function it called.

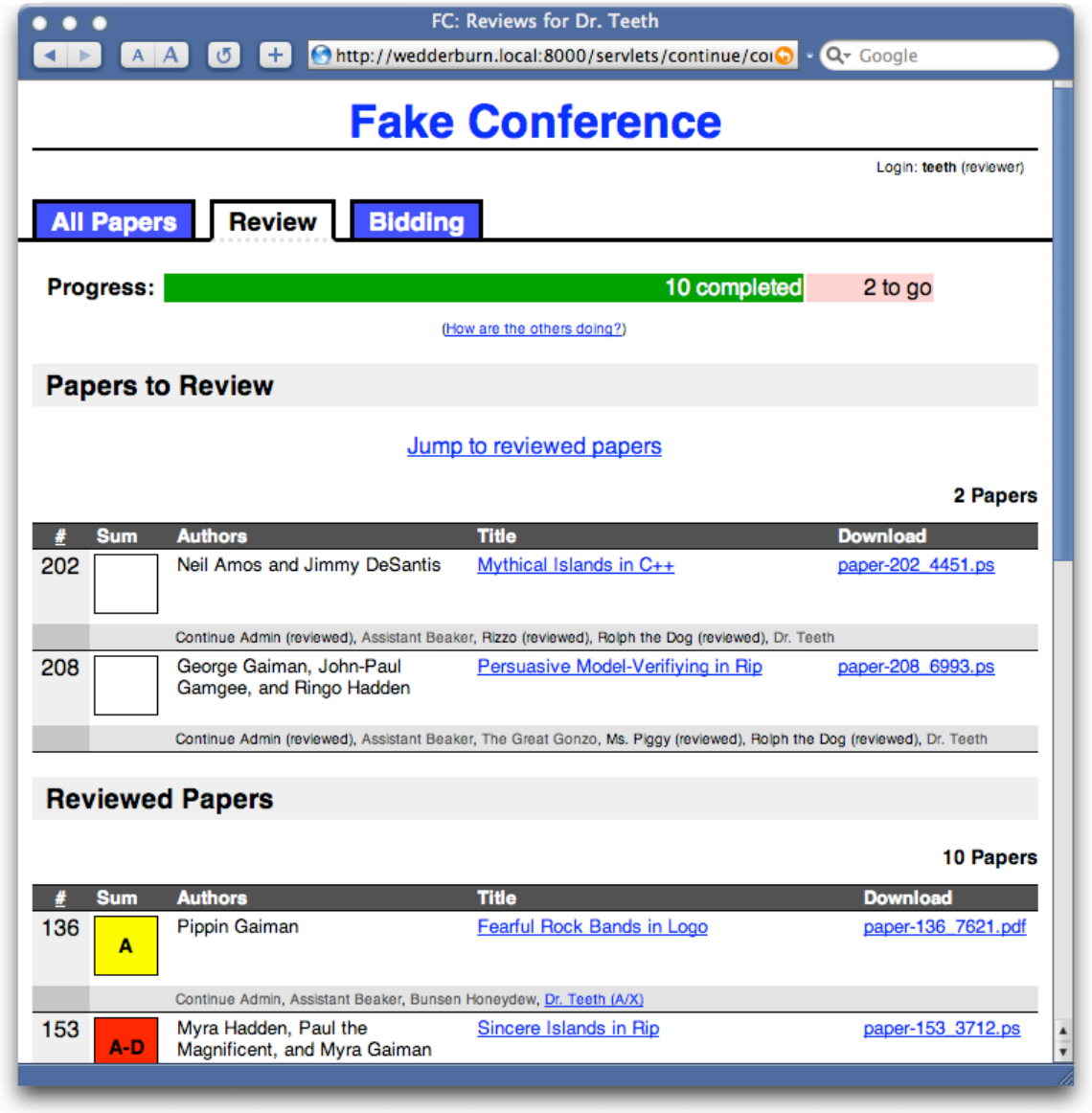


Figure 10. CONTINUE's interface for a sample conference, from a reviewer's perspective.

```
(define (show-reviews-list)
  (define request
    (send/suspend
     (lambda (k-url)
       `(html ...
              ,@(map (lambda (paper)
                       `(...
                         (a ([href ,(format " ~a?paper= ~a"
                                            k-url
                                            (paper-number paper))])
                            ,(paper-title paper))
                         ...))
                     (all-papers))
              ...))))
  (show-paper-reviews
   (string→number
    (extract-binding/single `paper (request-bindings request)))))
```

Figure 11. Pattern based on *send/suspend* in early versions of CONTINUE.

The discussion above shows how to add tabs to one particular page generator (showing the list of reviews). The same tabs must, however, also appear on other pages, such as those showing the list of papers. To avoid copying code, the developer must therefore abstract over the generation of tabs, and over their dispatching. Doing this in the natural manner, however, forces generation and dispatching to reside in different functions, which must now jointly maintain an invariant. Furthermore, the Web forces all URL query strings, and therefore request bindings, to reside in the same namespace. It is not hard to imagine that two separate interface elements (or the same element included twice) could use the same key to store their data in the query string. The resulting collision during dispatch would likely cause unexpected behavior. Furthermore, it is difficult for the developer who composed these two elements onto the same page to detect this problem ahead of time without studying the respective implementations, i.e., the developer cannot treat the abstractions as reusable black-boxes.

4. Demultiplexing Continuations

Finding a solution to the problems discussed in section 3.2 requires re-examining the relationship between *send/suspend*'s model of a Web page and our conceptual model of the *show-reviews-list* page. A careful comparison shows that the problem is an impedance mismatch. The procedure *send/suspend* provides a single URL to a given page-generation function, with the implication that the generated page has only one continuation to resume.

```
(define (show-reviews-list)
  (define request
    (send/suspend
     (lambda (k-url)
       `(html ...
              (ul ([id "tabs"])
                  (li (a ([href ,(format " ~a?tab=all" k-url)])
                         "All Papers"))
                  (li (a ([href ,(format " ~a?tab=review" k-url)])
                         "Review"))
                  (li (a ([href ,(format " ~a?tab=bidding" k-url)])
                         "Bidding")))
              ...
              ,@(map (lambda (paper)
                       `(...
                         (a ([href ,(format " ~a?paper= ~a"
                                            k-url
                                            (paper-number paper))])
                            ,(paper-title paper))
                         ...))
                     (all-papers))
              ...))))
  (define bindings (request-bindings request))
  (cond
    [(exists-binding? 'tab bindings)
     (case (extract-binding/single 'tab bindings)
       [("all") (show-papers-list)]
       [("review") (show-reviews-list)]
       [("bidding") (show-bidding-list)])]
    [(exists-binding? 'paper bindings)
     (show-paper-reviews (string→number (extract-binding/single 'paper bindings)))]
    [else (error ...)]))
```

Figure 12. Extending *show-reviews-list* with tabs.

This is true at the implementation level. Conceptually, however, pages generated by procedures such as *show-reviews-list* refer to several "virtual" continuations—in the example, one for each paper and tab—and the user has the power to choose which one the servlet should "resume". To reflect this, such pages must refer to several, distinct URLs.

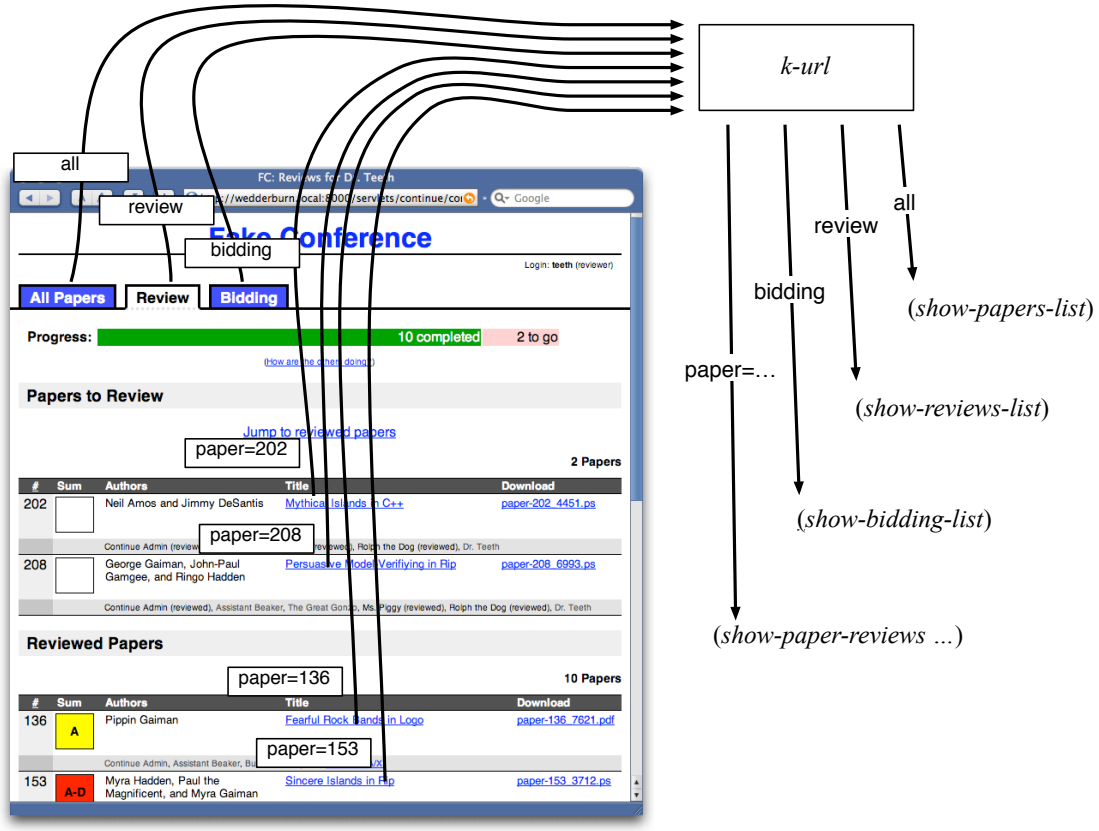


Figure 13. Multiplexing and demultiplexing requests.

4.1. A New Interaction Primitive

Our solution is a new interaction primitive, dubbed *send/suspend/dispatch* [20], whose implementation we discuss in section 4.2. Conceptually, this new primitive does automatically what the developer earlier did by hand. It maintains a mapping from keys to procedures representing the rest of the computation; it then generates a fresh key upon demand. When the user submits a request, the primitive intercepts the request, extracts the key, and invokes the procedure associated with that key.

To make this process transparent, *send/suspend/dispatch* has a different interface than *send/suspend*:

;; *send/suspend/dispatch* : ((HTTP-Request → any) → URL → X-Expression) → any

Whereas *send/suspend* invokes its page-generation function with a single unique URL, *send/suspend/dispatch* invokes its page-generation function with a function that gener-

ates unique URLs on the fly. This URL-generating function, which is the boxed portion of the interface above and is conventionally called *embed/url*, consumes a function of one argument and produces a unique URL. When the user requests one of these URLs, the associated function is invoked with the browser's request datum as its argument.

Figure 14 presents the rewritten version of *show-reviews-list*, along with a single-function implementation of tabs. In Figure 12, the developer had to generate a distinguished URL in the page generator:

```
‘(... (li (a ([href ,(format "~a?tab=all" k-url)])
             "All Papers")) ...)
```

This embedded name, "all", was then used to dispatch upon receiving the user's response:

```
[("all") (show-papers-list)]
```

As a result, the developer had to (a) devise a unique name, (b) manually synthesize a URL, and (c) maintain an invariant between two different fragments of code. While some of these problems can be ameliorated by helpful abstractions, the third problem remains. In contrast, as Figure 14 shows, *send/suspend/dispatch* permits the two code fragments to be combined:

```
‘(... (li (a ([href ,(embed/url (lambda _ (show-papers-list)))])
             "All Papers")) ...)
```

Notice that *embed/url* is just a procedure, so it can be passed as an argument to other procedures, such as *tabs*. This shows how it is possible to define abstractions for visual elements and yet incorporate them into a single page.

The argument supplied to *embed/url* is a function that consumes the request and runs an arbitrary computation. Because the paper list doesn't depend on the details of the request, only which tab or link was clicked, the procedure supplied by the developer ignores its argument. (This supplied procedure does, however, needs to be a genuine closure: in *show-reviews-list* it closes over *paper*, the iteration variable mapping over paper identifiers.) The use of *send/suspend/dispatch* avoids the need to serialize the paper number in the query string, and also eliminates the need to invent names for the tags.[16]

While this example uses *embed/url* to generate anchor elements (links) only, it is certainly possible to use it to generate a URL in any other valid context, such as in the `action` field of a form. Unfortunately, because the action is associated with a form, not with a button, *send/suspend/dispatch* cannot distinguish between different form buttons: this must still be done manually in the style of figure 12. Therefore, some users prefer to use links rather than buttons to represent different choices for the next page.

[16] The former is important because it greatly reduces the likelihood that a user (in the case of CONTINUE, a reviewer) might forge a URL and thereby gain access to a document that he should not have seen (such as reviews for a paper with which he has a conflict of interest). While a general access-control mechanism is necessary to prevent such attacks, and we have been incorporating one [14], an obscure reference mechanism provides a baseline of protection.

```
(define (tabs embed/url)
  `(ul ([id "tabs"])
       (li (a ([href ,(embed/url (lambda _ (show-papers-list)))])
              "All Papers"))
       (li (a ([href ,(embed/url (lambda _ (show-reviews-list)))])
              "Review"))
       (li (a ([href ,(embed/url (lambda _ (show-bidding-list)))])
              "Bidding"))))

(define (show-reviews-list)
  (send/suspend/dispatch
   (lambda (embed/url)
     `(html
       ...
       ,(tabs embed/url)
       ...
       ,@(map (lambda (paper)
                `(...
                  (a ([href ,(embed/url
                              (lambda _
                                (show-paper-reviews (paper-number paper))))])
                     ,(paper-title paper))
                  ...))
              (all-papers))
       ...))))
```

Figure 14. show-reviews-list written in terms of *send/suspend/dispatch*.

A keen reader may notice that the *tabs* function from figure 14 contains code reminiscent of continuation-passing style, a practice that we spent the beginning of this paper denouncing. While the argument to *embed/url* is a closure representation of a continuation, this represents a purely local transformation, unlike CPS, which is global. The vast majority of the servlet code is still written in direct style. Furthermore, this style of embedding code is strongly reminiscent of the *page-centric* Web design methodology espoused by languages such as ColdFusion or Java Server Pages, where a Web page's source mixes presentation with control behavior. While page-centric methods suffer from problems of scale and maintenance, if at all such a style is natural, *send/suspend/dispatch* offers an approximation of it.

```
(define (send/suspend/dispatch response-generator)
  (let/ec escape
    (define (callback→url callback)
      (let/ec return-url
        (escape (callback (send/suspend return-url)))))
    (send/back (response-generator callback→url))))
```

Figure 15. Implementation of *send/suspend/dispatch*.

4.2. Implementation

Our initial instinct might be to implement *send/suspend/dispatch* by creating a hash-table to maintain the keys, generating fresh entries in the hash-table upon demand, synthesizing a URL that refers to the fresh key, looking up the hash-table on receipt of a request, and dispatching to the indexed entry of the hash-table. This is the solution we presented in our prior work [20].

Fortunately, there is a far simpler implementation of the same primitive, which we present in figure 15. This employs the Scheme form **let/ec**, which captures an *escaping* continuation and binds it to the name specified in the first sub-form (here, respectively, *escape* and *return-url*). Escaping continuations are active only during the dynamic extent of the expression that created them; they cannot be used to re-enter that extent. As such, escaping continuations are analogous to conventional exceptions (in behavior and in cost).

The implementation of *send/suspend/dispatch* in figure 15 works as follows.

- After establishing *escape*, it defines the procedure *callback→url*. This procedure is passed to *response-generator*, and is thus the value conventionally called *embed/url*.
- Suppose *response-generator* invokes *embed/url* on a callback function. The body of *callback→url* generates a fresh URL using *send/suspend*, then returns this URL (via *return-url*) to the context that invoked *embed/url*, typically for association with a Web page link.
- When a user visits one of these URLs, control passes to the context surrounding (*send/suspend return-url*), thereby invoking the callback procedure. If the callback ever completes, control *escape*s to the context of applying *send/suspend/dispatch*.

In sum, this implementation recognizes that *send/suspend* already has the power to (a) generate unique URLs, and (b) dispatch on responses; furthermore, it exploits the fact that *send/suspend* is built atop first-class continuations.

5. Web Data and Programming Style

Let us revisit the sequence of Web interactions presented at the beginning of section 1. The application can easily exhibit the erroneous behavior if the developer stored the information about the current book in shared state such as a session object. Viewing book B would update this shared state, and making the purchase from the page for book A would still buy a copy of book B. In contrast, storing the information about the book in a lexical binding would lead to two distinct copies, one per page, because continuations close over their environment. As a result, buying from the page for book A would result in a copy of book A.

This example partially illustrates a taxonomy of data on the Web:

Page-local data is meta-information about what a specific HTML page is displaying or accumulated information from a transaction of which that page is a part. For a page about a book, one of its page-local data might be the book's ISBN number. The page-local data for the confirmation step of a shopping site would include the items being purchased, shipping, and billing information. Page-local data matches the semantics of hidden HTML form fields.

Session data are tied to a particular user and persist across all the pages that a user visits with the same browser in some time period. The canonical session data example is a shopping cart, since its state persists no matter what pages a user visits or re-visits on a site. Though a session is unique to a user, a user could have multiple sessions if, for example, that user is using several computers, or different browsers on the same computer. Sessions have reasonable persistence, usually at least until the user quits the browser, but no guarantee of permanence. Sessions match the semantics of HTTP cookies, and are usually implemented in terms of them.[17]

Global data are shared among all invocations of a Web application. Continuing the bookstore example, the set of books and their availability would be global data. So would the set of past transactions. Global data can be kept in memory, but are commonly stored in a database.

Because the CGI specification has no native support for data storage, developers using the CGI protocol typically use hidden form fields, cookies, and a database for these three kinds of data, respectively. These three options are always available to Web application developers, regardless of platform, but the difficulty and security risks inherent in serialization—which is necessary to employ form fields and cookies—leads developers to use alternate mechanisms when possible.

Not all platforms support all three categories of data natively (see table IV, which refers to the case of a single server). Consequently, developers sometimes use the mechanisms that are most convenient, even if these do not neatly fit the desired behavior of the data.

[17] Some Web application platforms such as PHP offer other techniques to maintain sessions, such as URL rewriting, but these are generally fallback options for when a user has turned off cookies in the browser.

Table IV. The taxonomy of Web data, and its implementation on three platforms.

	HTML/HTTP	Java Servlets	PLT Scheme Web Server
Page-Local	URLs and hidden form fields	hidden form fields	the environment
Session	cookies	`HttpSession` objects	fluid scope
Global	external database	`static` member fields	the store

For instance, servlet developers using Java often employ session objects because they are supported well by the Java servlet API, even if the data they are trying to represent are page-local, for which the developer must employ what is effectively an extra-lingual technique (of modifying the page generating code, which may not even be under that developer's control). This results in the kind of error presented earlier in this section.

In general, session objects engender a *stateful* programming style, in which variables and objects are mutated over the course of a users's interaction with the application. For a traditional console-based or GUI application, this paradigm is reasonable because the user's state and the application's state are inexorably linked. The user has no external means of affecting the application's control flow.

As we saw in section 2.4, however, the user has access to silent operations that affect the client context but are not immediately broadcast to the server. The operations are so prevalent and intrinsic to how users interact with the Web that Web applications must behave reasonably in the face of them or risk confusing users with unexpected or erroneous behavior. Therefore, session objects can only be used safely when the servlet API provides a mechanism to detect a silent operation and correspondingly roll back the state of all objects to previously recorded states.[18]

In the PLT Scheme Web Server, page-local data correspond precisely to information stored in the current environment, over which lexically-scoped continuations are automatically closed. As a result, programming in a functional style automatically leads to data obeying the page-local behavior, without the need for further intervention from the developer. Global information, similarly, corresponds precisely to data in the store, so using imperative operations such as **set!** automatically ensures there is a single, global copy of the datum that the application manipulates (this follows because the continuation does not close over the store). Using lexical scope to introduce an identifier, and then fluid scope to modify it, translates to the equivalent of a session. If data need to persist across sessions, developers can access the cookie functionality of PLT Scheme (which is similar to the APIs provided for this purpose in other Web programming libraries).

[18] The Squeak-based Seaside [41] platform has a mechanism to do this, as does Apache's Cocoon [44].

6. Related Work

6.1. Content Validation

Many research projects have covered various aspects of static validation, such as ensuring that the generated document or the accesses to form fields are valid; representative examples are those by Wallace and Runciman [48] and by Meijer and van Velzen [30]. These are all essentially orthogonal to the concerns addressed by the PLT Scheme Web Server; since many of these are defined for functional languages, it should be feasible to port them to Scheme. In practice, we have found that form field access problems tend to be caught quickly while testing, and we have found browsers tend to be forgiving with HTML (the advent of cascading style-sheets greatly simplifies the HTML, further reducing the need for validation). The server would, nevertheless, benefit from some of these tools, and indeed an early version contained static validation support [25].

6.2. Other Languages with Powerful Interaction Primitives

Seaside [41] is a Smalltalk-based Web development framework with continuation support. Seaside models a Web page as a tree of nested component objects, each of which can independently render itself and respond to user interaction. A component can use the `call` method to suspend its own processing and replace itself with another component, which appears in its place on the page. The replacing component may use its `answer` method to remove itself, resuming the original component and passing it data. As with *send/suspend/dispatch*, Seaside allows Web applications to associate blocks of code with links, which are executed when the links are clicked. Applications do not, however, have access to the URLs generated for these links, as they do with *send/suspend/dispatch*, and therefore cannot re-use them on a page to conserve resources, modify them, employ them in generated JavaScript, or include them in verification emails. Seaside also supports attaching blocks of code to form input elements and submit buttons, which get executed automatically when the associated form is submitted. Though Seaside's object-oriented approach renders it more susceptible to interaction errors and other inappropriate mutations, it overcomes this with a mechanism to "backtrack" the state of components if the user clicks the back button (by sending a pointer into a transaction log that tracks those variables the programmer has marked as "backtrackable").

While Seaside's component-based approach is good for modularizing complex pages, it suffers from the limitation that a single component cannot interrupt its computation to send a page to the user without passing control to another component using the `call` method. Furthermore, Seaside's request-processing strategy requires that a component define a method to return a list of all subcomponents it might render, separate from the method that actually chooses and renders those components, thereby inhibiting reuse.

The <`bigwig`> system [5] is a domain-specific language for Web development that includes statically-checked, higher-order templates and a "session-based" model of a sequence of Web interactions as a single, coherent flow-of-control. Through flow-based analysis,

<bigwig> programs are statically checked for template consistency—for example, that a template includes an expected form field name—and HTML validity. <bigwig> is implemented with a custom Apache module that allocates and maintains a server process for each user session.

The <bigwig> system has no analogue to `send/suspend/dispatch`. It instead provides a "document reference" operator that allows one template to contain a hyperlink to another, forming a—potentially cyclic—"document cluster" of the templates. These clusters must be comprised of templates in the same scope that have identical form elements. Therefore, although its support for form-based interaction is strong, <bigwig> is unsuitable for content-focused Web applications.

Additionally, Web applications using <bigwig> do not conform to users' expectations of the back button. Because <bigwig> cannot model backtracking in its linear, process-based model of interactions, it disallows such operations. It accomplishes this by always redirecting the user's browser to a consistent URL that identifies the session, while the content at that URL changes dynamically. The session only generates a single entry in the browser's history, so if the user clicks the back button, the browser goes to the page that immediately preceded the start of the session. We believe this solution is unsatisfactory as it does not correspond to a user's expectations of the back button's behavior.

JWIG [8] is the Java-based successor to <bigwig>. It is compatible with standard Java and is implemented with a combination of Java libraries and a preprocessor that converts the JWIG language extensions into standard Java and performs the static verifications.

WASH/CGI [47] is a comprehensive Web application framework for Haskell whose focus is to abstract away the creation and processing of HTML forms. Using Haskell's type system, WASH/CGI can determine the expected type for form inputs and automatically validate them, re-displaying the form with appropriate error messages if necessary. It provides similar pure-functional, type wrappers around cookies and server-side persistent state. WASH/CGI and PLT Scheme servlets share a similar syntactic structure, allowing arbitrarily placement of user interaction points. Instead of relying on continuations, however, WASH/CGI replays the application from the beginning on each interaction. This works in a purely functional setting, like Haskell, but with the cost of duplicate computation. Another difference is that applications cannot maintain environment state when users interact through hyperlinks, without complicated JavaScript that does not scale to URL clients such as in our email verification example.

Apache's Cocoon framework [44] was developed as part of a project to provide online documentation for the Apache Software Foundation's Java libraries. Owing to this history, Cocoon's page generation is built around the concept of a "pipeline," where data, encoded as an XML document, passes through several transformations (which can be described using Java or XSLT) before it is presented to the user as HTML, XML, or another format. One of Cocoon's unique features among Java frameworks is its support for continuation-based Web programming. Cocoon allows developers to write controller logic in JavaScript, which it interprets with an embedded version of the Mozilla Foundation's Rhino JavaScript interpreter. Cocoon's Flowscript JavaScript API provides a `sendPageAndWait` method that is analogous to *send/suspend*. Although the JavaScript code has full access to Java classes

and objects, Cocoon is limited by the inability to capture a continuation from Java. Once a JavaScript controller calls `sendPageAndWait` to invoke a pipeline to generate a page, the Web application cannot capture any continuations until the user responds to the page and control passes back into JavaScript. Since the view elements of Cocoon—the pipelines—cannot dynamically create continuations to tie to links, Cocoon has no analogue to *send/suspend/dispatch*.

Struts [46] imposes a model-view-controller architecture to Java servlet programming. It provides mechanisms for routing HTTP requests to application components in the controller layer, and connecting those components to appropriate model Java Beans and views. Views can be written in JSP or any other HTML-generating system, and Struts provides utilities for converting between Java Beans and HTML forms, with appropriate validation and error reporting. Struts gives developers the full capabilities of Java servlets. Nevertheless, Struts is based on a session-state model of Web applications and does not provide a mechanism for state with environment semantics. This makes it very susceptible to interaction errors.

PS3I is a Scheme-based Web servlet framework with continuation support. PS3I is described by Queinnec [39] as a multi-thread interpreter embedded in a Java Web server. Unfortunately, it implements only a limited subset of the Scheme language (one that is "nearly R^4RS-compliant", in the author's words). As a result, it is of limited practical utility and, in fact, the author notes that he has used the PLT Scheme Web Server in the most recent version of the project that motivated PS3I. It is likely that a re-implementation with SISC [31] as a base would be profitable for developers requiring embedding in Java.

The MAWL project [27] was an early effort to create a statically-typed domain-specific language, intended for embedding in host languages, for writing Web applications and avoiding common errors. At the time, the most common errors were of faulty HTML generation and incorrectly carrying of state from one page to the next. MAWL solves the first problem by embedding program code in HTML in a style similar to languages like PHP and ColdFusion. In solving the second problem, MAWL effectively implements a continuation-based approach by automatically supplying the `action` attribute of forms. Unfortunately, MAWL offers no primitives such as *send/forward* for managing continuations; it does not have a facility for demultiplexing continuations; and, it does not fully implement the equivalent of *send/suspend*. This last restriction is imposed by disallowing access to continuation URLs, which inhibits applications such as the one presented in section 3.1. Finally, each service session corresponds to a single thread that waits for the most recent page to return, which potentially makes the system vulnerable to errors induced by user interactions.

6.3. Performance and Resource Management

The performance problems of the CGI interface has led others to develop faster alternatives [34, 43]. In fact, one of the driving motivations behind the Microsoft .NET initiative appears to be the need to improve Web server performance, partially by eliminating process boundaries.[19] Apache provides a module interface that allows developers to link code into the server that, among other things, generates content dynamically for given URLs. However,

[19] Jim Miller, Microsoft .NET Common Language Runtime team, personal communication, 2000-10-20.

circumventing the underlying operating system's protection mechanisms without providing an alternative within the server opens the process for catastrophic failures.

FastCGI [34] provides a safer alternative by placing each servlet in its own process. Unlike traditional CGI, FastCGI processes handle multiple requests, avoiding the process creation overhead. The use of a separate process, however, generates bi-directional inter-process communication cost, and introduces coherence problems in the presence of state.

IO-Lite [36] demonstrates the performance advantages to programs that are modified to share immutable buffers instead of copying mutable buffers across protection domains. Since the underlying memory model of PLT Scheme provides safety and immutable string buffers, our server automatically provides this memory model without the need to alter programming styles or APIs.

The problem of managing resources in long-running applications has been identified before in the Apache module system [43], in work on resource containers [4], and elsewhere. The Apache system provides a pool-based mechanism for freeing both memory and file descriptors en masse. Resource containers provide an API for separating the ownership of resources from traditional process boundaries. The custodians [16] in PLT Scheme provide similar functionality with a simpler API than those mentioned.

Like our server programs, Java servlets [9] are content generating code that runs in the same runtime system as the server. Passing information from one request to the processing of the next request cannot rely on storing information in instance variables since the servlet may be unloaded and re-loaded by the server. Passing values through static variables does not solve the problem either, since in some cases the server may instantiate multiple instances of the servlet on the same JVM or on different ones. Instead they provide session objects that assist the developer with the task of manually marshaling state into and out of re-written URLs, cookies, or secure socket layer connections.

The J-Server [40] runs atop Java extended with operating systems features. The J-Server team identified and addressed the server's need to prevent servlets from shutting down the entire server, while allowing the server to shutdown the servlets reliably. They too identified the need for servlets to communicate with each other, but their solution employs remote method invocation (which introduces both cost and coherence concerns) instead of shared lexical scope. Their work addresses issues of resource accounting and quality of service, which is outside the scope of this paper. Their version of Java lacks a powerful module system and first-class continuations.

One shortcoming with our server is the lack of support for marshalling continuations in the underlying virtual machine, unlike systems such as Kali Scheme [7]. This makes our server susceptible to outages. In contrast, a purely compiler-based approach [29] does not suffer from this shortcoming, making it more fault-tolerant. As we add continuation marshalling to PLT Scheme, however, this will cease to be a concern.

7. Conclusion

We have studied the construction and a particular use of the PLT Scheme Web Server. The server is built entirely as a Scheme application atop PLT Scheme, which provides primitives that enable the server to behave like a host operating system for Web applications. The implementation of PLT Scheme is sufficient for the server to match the performance of other servers in popular use.

The server has been used to build several practical applications, the most significant of which is probably CONTINUE, a conference paper manager that has been employed by several conferences. Building concrete applications atop the server has identified interesting patterns of use, as well as problems of expression that have helped us build better abstractions. These applications have opened several avenues for future work, relating to both performance and better linguistic expressiveness.

Acknowledgements

We thank Matthew Flatt for his superlative work on MzScheme. Numerous people have provided invaluable feedback on the server, including Eli Barzilay, Ryan Culpepper, Robby Findler, Dan Licata, Matt Jadud, Jacob Matthews, Matthias Radestock, Andrey Skylar, Michael Sperber, Dave Tucker, Anton van Straaten, and Noel Welsh. We also thank the many other PLT Scheme users who have exercised the server and offered critiques. Many CONTINUE users have endured its defects and provided valuable feedback, particularly George Avrunin, Michael Ernst, Robby Findler, Matthew Flatt, Gopal Gupta, Joshua Guttman, Alan Mycroft, C.R. Ramakrishnan, Philip Wadler, and Andreas Zeller. Finally, we thank the anonymous reviewers for numerous editing suggestions, including proposals for structural changes, that greatly improved the presentation.

References

1. Amdahl, G. M.: 1967, 'Validity of the single-processor approach to achieving large scale computing capabilities'. In: *AFIPS Conference Proceedings*, Vol. 30. pp. 483–485.
2. Aron, M., D. Sanders, P. Druschel, and W. Zwaenepoel: 2000, 'Scalable Content-Aware Request Distribution in Cluster-based Network Servers'. In: *USENIX Annual Technical Conference*. pp. 323–336.
3. Banga, G. and P. Druschel: 1999, 'Measuring the Capacity of a Web Server Under Realistic Loads'. *World Wide Web* **2**(1–2), 69–83.
4. Banga, G., P. Druschel, and J. Mogul: 1999, 'Resource containers: A new facility for resource management in server systems'. In: *Symposium on Operating System Design and Implementation*. pp. 45–58.
5. Brabrand, C., A. Møller, and M. I. Schwartzbach: 2002, 'The <bigwig> project'. *ACM Transactions on Internet Technology* **2**(2), 79–114.
6. Bray, T., J. Paoli, and C. Sperberg-McQueen: 1998, 'Extensible Markup Language XML'. Technical report, World Wide Web Consortium. Version 1.0.

7. Cejtin, H., S. Jagannathan, and R. Kelsey: 1995, 'Higher-Order Distributed Objects'. *ACM Transactions on Programming Languages and Systems* **17**(5), 704–739.
8. Christensen, A. S., A. Møller, and M. I. Schwartzbach: 2003, 'Extending Java for high-level Web service construction'. *ACM Transactions on Programming Languages and Systems* **25**(6), 814–875.
9. Coward, D.: 2000, 'Java Servlet Specification Version 2.3'. `http://java.sun.com/products/servlet/index.html`.
10. Danvy, O.: 1994, 'Back to Direct Style'. *Science of Computer Programming* **22**(3), 183–195.
11. Felleisen, M.: 2003, 'Developing Interactive Web Programs'. In: J. Jeuring and S. Peyton Jones (eds.): *Advanced Functional Programming School*, No. 2638 in Springer Lecture Notes in Computer Science. Springer-Verlag, pp. 100–128.
12. Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen: 2002, 'DrScheme: A Programming Environment for Scheme'. *Journal of Functional Programming* **12**(2), 159–182.
13. Fischer, M. J.: 1972, 'Lambda calculus schemata'. *ACM SIGPLAN Notices* **7**(1), 104–109. In the ACM Conference on Proving Assertions about Programs.
14. Fisler, K., S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz: 2005, 'Verification and Change-Impact Analysis of Access-Control Policies'. In: *International Conference on Software Engineering*. pp. 196–205.
15. Flatt, M. and M. Felleisen: 1998, 'Cool Modules for HOT Languages'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 236–248.
16. Flatt, M., R. B. Findler, S. Krishnamurthi, and M. Felleisen: 1999, 'Programming Languages as Operating Systems (or, Revenge of the Son of the Lisp Machine)'. In: *ACM SIGPLAN International Conference on Functional Programming*. pp. 138–147.
17. Graham, P.: 2001, 'Lisp for Web-Based Applications'. `http://www.paulgraham.com/lwba.html`.
18. Graunke, P. T., S. Krishnamurthi, S. van der Hoeven, and M. Felleisen: 2001, 'Programming the Web with High-Level Programming Languages'. In: *European Symposium on Programming*. pp. 122–136.
19. Herman, D.: 2005, 'WebPostRedirectGet'. `http://schemecookbook.org/Cookbook/WebPostRedirectGet`.
20. Hopkins, P. W.: 2003, 'Enabling Complex UI in Web Applications with send/suspend/dispatch'. In: *Scheme Workshop*. pp. 53–58.
21. Hughes, J.: 2000, 'Generalising monads to arrows'. *Science of Computer Programming* **37**(1–3), 67–111.
22. Kiselyov, O.: 2002, 'SXML Specification'. *ACM SIGPLAN Notices* **37**(6), 52–58.
23. Krishnamurthi, S.: 2003, 'The CONTINUE Server'. In: *Symposium on the Practical Aspects of Declarative Languages*. pp. 2–16.
24. Krishnamurthi, S., R. B. Findler, P. Graunke, and M. Felleisen: 2006, 'Modeling Web Interactions and Errors'. In: D. Goldin, S. Smolka, and P. Wegner (eds.): *Interactive Computation: The New Paradigm*, Springer Lecture Notes in Computer Science. Springer-Verlag. To appear.
25. Krishnamurthi, S., K. E. Gray, and P. T. Graunke: 2000, 'Transformation-by-Example for XML'. In: *Symposium on the Practical Aspects of Declarative Languages*. pp. 249–262.
26. Kristol, D. and L. Montulli: 2000, 'HTTP State Management Mechanism'. IETF RFC 2965. `http://www.ietf.org/rfc/rfc2965.txt`.
27. Ladd, D. A. and J. C. Ramming: 1995, 'Programming the Web: An Application-Oriented Language for Hypermedia Service Programming'. In: *International World Wide Web Conference*.
28. Lawall, J. L. and D. P. Friedman: 1992, 'Towards Leakage Containment'. Technical Report TR-346, Indiana University, Bloomington, IN, USA.
29. Matthews, J., R. B. Findler, P. T. Graunke, S. Krishnamurthi, and M. Felleisen: 2004, 'Automatically Restructuring Programs for the Web'. *Automated Software Engineering Journal* **11**(4), 337–364.

30. Meijer, E. and D. van Velzen: 2000, 'Haskell Server Pages - Functional Programming and the Battle for the Middle Tier'. *Electronic Notes in Theoretical Computer Science* **41**(1). Proceedings of the ACM SIGPLAN Haskell Workshop.
31. Miller, S. G.: 2003, 'SISC: A Complete Scheme Interpreter in Java'. `http://sisc.sourceforge.net/sisc.pdf`.
32. NCSA, 'The Common Gateway Interface'. `http://hoohoo.ncsa.uiuc.edu/cgi/`.
33. Nørmark, K.: 2005, 'Web Programming in Scheme with LAML'. *Journal of Functional Programming* **15**(1), 53–65.
34. Open Market, Inc., 'FastCGI specification'. `http://www.fastcgi.com/`.
35. Pai, V. S., P. Druschel, and W. Zwaenepoel: 1999a, 'Flash: An efficient and portable Web server'. In: *USENIX Annual Technical Conference*. pp. 199–212.
36. Pai, V. S., P. Druschel, and W. Zwaenepoel: 1999b, 'IO-Lite: A Unified I/O Buffering and Caching System'. In: *Third Symposium on Operating Systems Design and Implementation*. pp. 15–28.
37. Pitman, K.: 1980, 'Special Forms in Lisp'. In: *Conference Record of the Lisp Conference*. pp. 179–187.
38. Queinnec, C.: 2000, 'The influence of browsers on evaluators or, continuations to program web servers'. In: *ACM SIGPLAN International Conference on Functional Programming*. pp. 23–33.
39. Queinnec, C.: 2004, 'Continuations and Web Servers'. *Higher-Order and Symbolic Computation* **17**(4), 277–295.
40. Spoonhower, D., G. Czajkowski, C. Hawblitzel, C.-C. Chang, D. Hu, and T. von Eicken: 1998, 'Design and Evaluation of an Extensible Web and Telephony Server based on the J-Kernel'. Technical Report TR98-1715, Department of Computer Science, Cornell University.
41. Stéphane Ducasse, Adrian Lienhard, L. R.: 2004, 'Seaside - A Multiple Control Flow Web Application Framework'. In: *European Smalltalk User Group - Research Track*.
42. Sun Microsystems, Inc.: 2003, 'JSR154 - Java™Servlet 2.4 Specification'. `http://jcp.org/aboutJava/communityprocess/final/jsr154/`.
43. Thau, R.: 1996, 'Design considerations for the Apache Server API'. In: *Fifth International World Wide Web Conference*. pp. 1113–1122.
44. The Apache Cocoon Project: 2005, 'User Documentation'. The Apache Software Foundation.
45. The Apache Software Foundation. `http://www.apache.org/`.
46. The Apache Struts Project: 2005, 'The Struts User's Guide'. The Apache Software Foundation.
47. Thiemann, P.: 2002, 'WASH/CGI: Server-Side Web Scripting with Sessions and Typed, Compositional Forms'. In: *Symposium on the Practical Aspects of Declarative Languages*. pp. 192–208.
48. Wallace, M. and C. Runciman: 1999, 'Haskell and XML: Generic Document Processing Combinators vs. Type-Based Translation'. In: *ACM SIGPLAN International Conference on Functional Programming*. pp. 148–159.
49. World Wide Web Consortium: 2002, 'XHTML 1.0: The Extensible HyperText Markup Language'. `http://www.w3.org/TR/xhtml1/`.