Programming the Web with High-Level Programming Languages

Paul Graunke, Department of Computer Science, Rice University
Shriram Krishnamurthi, Department of Computer Science, Brown University
Steve Van Der Hoeven, ESSI, Université de Nice
Matthias Felleisen, Department of Computer Science, Rice University

Abstract. Many modern programs provide operating system-style services to extension modules. A Web server, for instance, behaves like a simple OS kernel. It invokes programs that dynamically generate Web pages and manages their resource consumption. Most Web servers, however, rely on conventional operating systems to provide these services. As a result, the solutions are inefficient, and impose a serious overhead on the programmer of dynamic extensions.

In this paper, we show that a Web server implemented in a suitably extended high-level programming language overcomes all these problems. First, building a server in such a language is straightforward. Second, the server delivers static content at performance levels comparable to a conventional server. Third, the Web server delivers dynamic content at a much higher rate than a conventional server, which is important because a significant portion of Web content is now dynamically generated. Finally, the server provides programming mechanisms for the dynamic generation of Web content that are difficult to support in a conventional server architecture.

1 Web Servers and High-Level Operating Systems

A Web server provides operating system-style services. Like an operating system, a server runs programs (e.g., CGI scripts). Like an operating system, a server protects these programs from each other. And, like an operating system, a server manages resources (e.g., network connections) for the programs it runs.

Some existing Web servers rely on the underlying operating system to implement these services. Others fail to provide services due to shortcomings of the implementation languages. In this paper, we show that implementing a Web server in a suitably extended functional programming language is straightforward and satisfies three major properties. First, the server delivers *static* content at a performance level comparable to a conventional server. Second, the Web server delivers *dynamic* content at five times the rate of a conventional server. Considering the explosive growth of dynamically created Web pages [7], this performance improvement is important. Finally, our server provides programming mechanisms for the dynamic generation of Web content that are difficult to support in a conventional server architecture.

The basis of our experiment is MrEd [11], an extension of Scheme [15]. The implementation of the server heavily exploits four extensions: first-class *modules*, which help structure the server and represent server programs; preemptive *threads*; which are needed to execute server programs; *custodians*, which manage the resource consumption of server programs; and *parameters*, which control stateful attributes of threads. The server programs also rely on Scheme's capabilities for manipulating continuations as first-class values. The paper shows which role each construct plays in the construction of the server.

The following section is a brief introduction to MrEd. Section 3 explains the core of our server implementation. In section 4 we show how the server can be extended to support Scheme CGI scripts and illustrate how programming in the extended Scheme language facilitates the implementation of scripts. Sections 3 and 4 also present performance results. Section 5 discusses related work. The final section summarizes our ideas and presents areas for future work.

2 MrEd: a High-Level Operating System

MrEd [11] is a safe implementation of Scheme [15]; it is one of the fastest existing Scheme interpreters. Following the tradition of functional languages, a Scheme program specifies a computation in terms of values and legitimate primitive operations on values (creation, selection, mutation, predicative tests). The implementation of the server exploits traditional functional language features, such as closures and standard data structures, and also Scheme's ability to capture and restore continuations, possibly multiple times.

MrEd extends Scheme with structures, exceptions, and modules. The module system [10] permits programmers to specify *atomic units* and *compound units*. An atomic unit is a closed collection of definitions. Each unit has an import and an export signature. The import signature specifies what names the module expects as imports; the export signature specifies which of the locally defined names will become visible to the rest of the world. Units are first-class values. There are two operations on unit values: invocation and linking. A unit is invoked via the **invoke-unit/sig** special form, which must supply the relevant imports from the lexical scope. MrEd permits units to be loaded and invoked at run-time. A unit is linked—or *compounded*—via the **compound-unit/sig** mechanism. Programmers compound units by specifying a (possibly cyclic) graph of connections among units, including references to the import signature; the result is a unit.

The extended language also supports the creation of threads and thread synchronization. Figure 1 specifies the relevant primitives. Threads are created from 0-ary procedures (thunks); they synchronize via counting semaphores. For communication between parent and child threads, however, synchronization via semaphores is too complex. For this purpose, MrEd provides (thread) *parameters*. The form

(**parameterize** ([*parameter1 value1*]...) *body1* ...)

sets *parameter1* to *value1* for the dynamic extent of the computation *body1* ...; when this computation ends, the parameter is reset to its original value. New

tcp-listen : Nat [Nat] $\rightarrow$ Tcp-listener
;; reserves a port to accept connections, optionally specifying the
;; maximum number of clients that may wait for a connection

tcp-accept : Tcp-listener $\rightarrow^*$ Input-port Output-port
;; creates I/O ports for a connection request via the listener

thread : ($\rightarrow$ Void) $\rightarrow$ Thread
;; spawns a thunk as a thread

make-semaphore : Nat $\rightarrow$ Semaphore
;; creates a semaphore with specified number of tokens

semaphore-post : Semaphore $\rightarrow$ Void
;; posts a semaphore and releases waiting threads

semaphore-wait : Semaphore $\rightarrow$ Void
;; waits (and possibly suspends) for a semaphore

make-custodian : $\rightarrow$ Custodian
;; creates a custodian

custodian-shutdown-all : Custodian $\rightarrow$ Void
;; shuts down all threads in custodian and reclaims all resources

Fig. 1. MrEd's TCP, thread and custodian primitives

threads inherit copies of their parent's parameter bindings, though the parameter values themselves are not copied. That is, when a child sets a parameter, it does not affect a parent; when it mutates the state of a parameter, the change is globally visible. The server deals with only two of MrEd's standard parameters: *current-custodian* and *exit-handler*. The default *exit-handler* halts the entire runtime system. Setting this parameter to another function can cause conditions that would normally exit to raise an exception or perform clean up operations.

Finally, MrEd provides a mechanism for managing resources, such as threads (with associated parameter bindings), TCP listeners, file ports, and so on. When a resource is allocated, it is placed in the care of the current *custodian*, the value of the *current-custodian* parameter. Figure 1 specifies the only relevant operation on custodians: *custodian-shutdown-all*. It accepts a custodian and reaps the associated resources: it kills the threads in its custody, closes the ports, reclaims the TCP listeners, and recursively shuts down all child custodians.

3 Serving Static Content

A basic web server satisfies HTTP requests by reading Web pages from files. High-level languages ease the implementation of such a server, while retaining efficiency comparable to widely used servers. The first subsection explains the core of our server implementation. The second subsection compares performance figures of our server to Apache [2], a widely-used, commercially-deployed server.

3.1 Implementation of the Web Server's Core

The core of a web server is a wait-serve loop. It waits for requests on a particular TCP port. For each request, it creates a thread that serves the request. Then the server recurs:[1]

```
;; server-loop : Tcp-listener → Void
(define (server-loop listener)
  (let-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)
  (let-values ([(meth url-string major-version minor-version)
                (read-request ip op)])
    (let* ([headers (read-headers ip op)]
           [url (string→url url-string)]
           [host (find-host (url-host url) headers)])
      (dispatch meth host port url headers ip op))))

;; read-request : Input-port Output-port →* Symbol String String String
;; to read a request from ip, to parse it, and to determine the
;; request method (get, put), URL, and protocol versions
;; effect: raises an exception and closes the ports, if parsing fails
(define (read-request ip op) ...)
```

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. In either case, it closes the ports before returning.[2]

3.2 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. More precisely, we would like our server to serve content from files at about the same rate as Apache [2]. We believed that this goal was within

[1] **let-values** binds names to the values returned by multiple-valued computations such as *tcp-accept*, which returns input and output ports.

[2] The server may actually loop to handle multiple requests per connection. Our paper does not explore this possibility further.

	Connections/Second								
	1kB file			10kB file			100kB file		
Clients	MrEd	Apache	Ratio	MrEd	Apache	Ratio	MrEd	Apache	Ratio
2	967.5	1557.9	62.1%	655.1	771.6	84.9%	105.2	113.2	92.9%
4	986.7	1623.4	60.8%	772.0	1084.4	71.2%	110.2	115.7	95.2%
8	997.9	1607.0	62.1%	752.9	1099.0	68.5%	116.0	115.8	100.2%
16	982.8	1597.0	61.5%	782.6	1101.3	71.1%	116.5	116.1	100.3%
32	923.8	1551.0	59.6%	760.7	1104.0	68.9%	116.7	116.3	100.3%
64	917.6	1577.2	58.2%	787.1	1093.0	72.0%	115.1	116.5	98.8%
128	946.3	1547.8	61.1%	769.4	1104.1	69.7%	116.7	116.5	100.2%

Ratio = MrEd/Apache

The client and server software each ran on an AMD Athlon 800MHz processor with 192 Mbytes of memory, running FreeBSD 4.1.1-STABLE, connected by a standard 100 Mbit/s Ethernet connection.

Fig. 2. Performance for static content server

reach because most of the computational work involves parsing HTTP requests, reading data from disk, and copying bytes to a (network) port.[3]

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; 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 because the total response time would include an extra disk access component that would be similar for both servers.

The results in figure 2 show that we have essentially achieved our goal. The results were obtained using the S-client measuring technology [5]. For the historically most common case [4]—files between six and thirteen kB—our server performs at a rate of 60% to 80% of Apache. For larger files, which are now more common due to increased uses of multimedia documents, the two servers perform at the same rate. In particular, for one and ten kB files, more than four pending 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.

4 Dynamic Content Generation

Over the past few years, the Web's content has become increasingly dynamic. *USA Today*, for instance, finds that as of the year 2000, more than half of the Web's content is generated dynamically [7]. Servers no longer retrieve plain files

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

from disk but use auxiliary programs to generate a document in response to a request. These Web programs often interact with the user and with databases. This section explains how small changes to the code of section 3 accommodate dynamic extensions, that the performance of the revised server is superior to that of Apache,[4] and that it supports a new programming paradigm that is highly useful in the context of dynamic Web content generation.

4.1 Simple Dynamic Content Generation

Since a single server satisfies different requests with different content generators, we implement a generator as a module that is dynamically invoked and linked into the server context. More specifically, a CGI program in our world is a unit:

```
(unit/sig () (import cgi^) ⟨def+exp⟩ ... ⟨exp⟩)
```

It exports nothing; it imports the names specified in the *cgi^* signature. The result of its final expression (and of the unit invocation) is an HTML page.[5]

Here is a trivial CGI program using **quasiquote** [22] to create an HTML page with **unquote** (a comma) allowing references to the *TITLE* definition.

```
(unit/sig () (import cgi^)
  (define TITLE "My first web page")
  `(html (head (title ,TITLE))
         (body
           (p (center ,TITLE))
           (p "Hello, World!"))))
```

The script defines a title and produces a simple Web page containing a message.

The imports of a content generator supply the request method, the URL, the optional headers, and the bindings:

```
(define-signature cgi^ (
  method     ; (union 'get 'post 'head)
  url        ; Url
  headers    ; (listof (cons Symbol String))
  bindings   ; (listof (cons Symbol String))
  ...))
```

To add dynamic content generation to our server, we modify the *dispatch* function from section 3 to redirect requests for URLs starting with "/cgi-bin/". More concretely, instead of responding with the contents of a file, *dispatch* loads a unit from the specified location in the file system. Before invoking the unit, the function installs a new *current-custodian* and a new *exit-handler* via a **parameterize** expression:

[4] Apache outperforms most other servers for CGI-based content generation [1].

[5] To be precise, it generates an X-expression, which is an S-expression representation of an XML document. The server handles other media types also; we do not discuss these in this paper.

```
;; in dispatch:
...
(if (cgi-url? url)
    (let ([cust (make-custodian)])
      (parameterize
          ([current-custodian cust]
           [exit-handler (lambda (x) (custodian-shutdown-all cust))])
        (let ([cgi-program (cached-load (url-path url))])
          (output-xhtml (invoke-unit/sig cgi-program cgi^)))))
    ...)
```

The newly installed custodian is shut down on termination of the CGI script. This halts child threads, closes ports, and reaps the script's resources. The new *exit-handler* is necessary so that erronious content generators shut down only the custodian instead of the entire server.

4.2 Content Generators are First-Class Values

Since units are first-class values in MrEd, the server can store content generators in a cache. Introducing a cache avoids some I/O overhead but, more importantly, it introduces new programming capabilities. In particular, a content generator can now maintain local state across invocations. Here is an example:

```
(let ([count 0])
  (unit/sig () (import cgi^)
    (set! count (add1 count))
    `(html (head (title "Testing Persistent State of Counter"))
           (body (p "This is a cgi generated web page.")
                 (p "The current count is " ,(number->string count))))))
```

This generator maintains a local count that is incremented each time the unit is invoked to satisfy an HTTP request. Its output is an HTML page that contains the current value of *count*.

4.3 Exchanging Values between Content Generators

In addition to maintaining persistent state across invocations, content generators may also need to interact with each other. Conventional servers force server programs to communicate via the file system or other mechanisms based on character streams. This requires marshaling and unmarshaling data, a complex and error prone process. In our server architecture, dynamic content generators can naturally exchange high-level forms of data through the common heap.

Our dynamic content generation model features a simple extension that permits multiple generators to be defined within a single lexical scope. The current unit of granularity in the implementation is a file. That is, one file may yield an expression that contains multiple generators. The expression may perform arbitrary operations to define and initialize the shared scope. To distinguish between the generators, the server's interface requires that the file return an association list of type

(listof (*cons* Symbol Content-generator))

For instance, a file contain two content generators:

```
(let* ([data-file ...]
       [lock (make-semaphore 1)])
  `((add . ,⟨Content-generator⟩₁)
    (delete . ,⟨Content-generator⟩₂)))
```

This yields two generators that share a lock to a common file. The distinguishing name in the association is treated as part of the URL in a CGI request.

4.4 Interactive Generation of Content

In a recent ICFP article [23], Christian Queinnec suggested that Web browsing in the presence of dynamic Web content generation can be understood as the process of capturing and resuming continuations.[6] For example, a user can bookmark a generated page that contains a form and (try to) complete it several times. This action corresponds to the resumption of the content generator's computation after the generation of the first form.

Ordinarily, programming this style of computation is a complex task. Each time the computation requires interaction (responses to queries, inspection of intermediate results, and so forth) from the user, the programmer must split the program into two fragments. The first generates the request for interaction, typically as a form whose processor is the remainder of the computation. The first program must store its intermediate results externally so the second program can access and use them. The staging and marshaling are cumbersome, error-prone, slow, and inhibit the reuse of existing non-CGI programs that are being refitted with Web-based interfaces. To support this common programming paradigm, our server links content generators to the three additional primitives in figure 3.

```
send/suspend : (Url → Html-page) → (list Method
                                         Url
                                         (listof (cons Symbol String))
                                         (listof (cons Symbol String)))
send/finish : Html-page → Void
adjust-timeout : Nat → Void
```

Fig. 3. Additional content generator primitives

The *send/suspend* function allows the content generator to send an HTML form to the client for further input. The function captures the continuation and

[6] This idea also appears in Hughes's paper on arrows [14].

suspends the computation of the content generator. When the user responds, the server resumes the continuation with four values: the request method, the URL, the optional headers, and the form bindings.

To implement this functionality, *send/suspend* consumes a function of one argument. This function, in turn, consumes a unique URL and generates a form whose action attribute refers to the URL. When the user submits the form, the suspended continuation is resumed. Consider figure 4, which presents a simple example of an interactive content generator. This script implements curried multiplication, asking for one number with one HTML page at a time. The two underlined expressions represent the intermediate stops where the script displays a page and waits for the next set of user inputs. Once both sets of inputs are available it produces a page with the product.

```
(unit/sig () (import cgi^)

  ;; get-input-w/-short-form : String → (String → Html-page)
  (define (get-input-w/-short-form which-one)
    (let-values ([(method url headers bindings)
                  (send/suspend
                    (lambda ( k-url )
                      `(html (head (title ,which-one " number"))
                         (body
                           (form ((method "post") (action , k-url ))
                             "Enter the " ,which-one " number:" nbsp
                             (input ((type "text") (name ,which-one)))
                             (input ((type "submit") (name "submit")))))))))])
      (extract which-one bindings)))

  ;; string-multiply : String String → String
  ...

  `(html (head (title "Product"))
     (body (p "The product is: "
              ,(string-multiply (get-input-w/-short-form "first")
                                (get-input-w/-short-form "second"))))))
```

Fig. 4. An interactive CGI program

In general, this paradigm produces programs that naturally correspond to the flow of information between client and server. These programs are easier to match to specifications, to validate, and to maintain. The paradigm also causes problems, however. The first problem, as Queinnec points out, concerns garbage collection of the suspended continuations. By invoking *send/suspend*, a content generator hands out a reference to its current continuation. Although these references to continuations are symbolic links in the form of unique URLs, they are nevertheless references to values in the server's heap. Without further restrictions, garbage collection cannot be based on reachability. To make matters worse, these continuations also hold on to resources, such as open files or TCP connections, which the server may need for other programs.

Giving the user the flexibility to bookmark intermediate continuations and explore various choices creates another problem. Once the program finishes interacting with the user, it records the final outcome by updating persistent state on the server. These updates must happen at most once to prevent catastrophes such as double billing or shipping too many items.

Based on this analysis, our server implements the following policy. When the content generator returns or calls the *send/finish* primitive, all the continuations associated with this generator computation are released. Generators that wish to keep the continuations active can suspend instead. When a user attempts to access a reclaimed continuation, a page directs them to restart the computation. Furthermore, each instance of a content generator has a predetermined lifetime after which continuations are disposed. Each use of a continuation updates this lifetime. A running continuation may also change this amount by calling *adjust-timeout*. This mechanism for shutting down the generator only works because the reaper and the content generator's thread share the same custodian. This illustrates why custodians, or resource management in general, cannot be identified with individual threads.

4.5 Implementing Interactive Generation of Content

The implementation of the interactive CGI policy is complicated by the (natural) MrEd restriction that capturing a continuation is local to a thread and that a continuation can only be resumed by its original thread. To comply with this restriction, *send/suspend* captures a continuation, stores it in a table indexed by the current content generator, and causes the thread to wait for input on a channel. When a new request shows up on the channel, the thread looks up the matching continuation in its table and resumes it with the request information in an appropriate manner. A typical continuation URL looks like this:

```
http://www/cgi-bin/send-test.ss;id38,k3-839800468
```

This URL has two principle pieces of information for resuming computation: the thread (`id38`) and the continuation itself (k3). The random number at the end serves as a password preventing other users from guessing continuation URLs.

The table for managing the continuations associated with a content generator actually has two tiers. The first tier associates instance identifiers for content generators with a channel and a continuation table. This continuation table associates continuation identifiers with continuations. Here is a rough sketch:

generator instance table	
instance-id	channel × continuations-table
⋮	⋮
instance-id	channel × continuations-table

continuations table	
continuation-id	continuation
⋮	⋮
continuation-id	continuation

When a thread processes a request to resume its generator's continuation, it looks up the content generator in one table, and extracts the channel and continuation table for that generator. The server then looks up the desired continuation in this second table and passes it along with the request information and the ports for the current connection.

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

4.6 Performance

We expect higher performance from our Web server than from conventional servers that use the Common Gateway Interface (CGI) [19]. A conventional server starts a separate OS process for each incoming request, creating a new address space, loading code, etc. Our server eliminates these costs by avoiding the use of OS process boundaries and by caching CGI programs.

Our experiments confirm that our server handles more connections per second than CGI programs written in C. For example, for the comparison in figure 5, we clock a C program's binary in CGI and FastCGI 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 sizes the network bandwidth becomes the dominant factor just as with static files.

The table also indicates that both the standard CGI implementation and our server scale much better relative to response size than FastCGI does. We conjecture that this is because FastCGI copies the response twice, and is thus much more sensitive to the response size. Of course, as computations become more intensive, the comparison becomes one of compilers and interpreters rather than of servers and their protocols. We are continuing to conduct experiments, and intend to present the additional performance measurements in a future edition of this paper.

4.7 Modularity of the Server

Web servers must not only be able to load web programs (e.g., CGI scripts) but also load new modules in order to extend their capabilities. For example, requir-

	CGI		FastCGI		MrEd Full		MrEd Lite	
Clients	1kB	10kB	1kB	10kB	1kB	10kB	1kB	10kB
8	161.1	158.7	742.7	551.6	766.5	665.9	851.4	742.6
16	157.6	156.9	728.8	547.2	759.6	659.3	847.3	727.9
32	153.4	153.1	720.7	544.4	733.8	627.4	837.8	721.4

MrEd Full is the server described in this paper. It includes the continuation reaper described at the end of section 4.4, whose implementation is currently quite inefficient. MrEd Lite disables this reaper (rendering *send/suspend* less usable), making its services more directly comparable to those of CGI and FastCGI.

Fig. 5. Performance for dynamic content generation

ing password authentication to access particular URLs affects serving content from files and from all dynamic content generators. In order to facilitate billing various groups hosted by the server, the administrator may find it helpful to produce separate log files for each client instead of a monolithic one. A flexibly structured server will split key tasks into separate modules, which can be replaced at link time with alternate implementations.

Apache's module system [25] allows the builder of the Web server to replace pieces of the server's response process, such as those outlined above, with with their own code. The builder installs structures with function pointers into a Chain of Command pattern [13]. Using this pattern provides the necessary extensibility but it imposes a complex protocol on the extension programmer, and it fails to provide static guarantees about program composition.

In contrast, our server is constructed in a completely modular fashion using the unit system [10]. This provides the flexibility of the Apache module system in a less ad hoc, more hierarchical manner. To replace part of how the server responds to requests, the server builder writes a compound unit that links the provided units and the replacement units together, forming an extended server. Naturally, the replacement units may link to the original units and delegate to them as desired.

Using units instead of dynamic protocols has several benefits. First, the server doesn't need to traverse chains of structures, checking for a module that wants to handle the request. Second, the newly linked server and all the replacement units are subject to the same level of static analysis as the original server. (Our soft-typing tool [9] revealed several easily corrected premature end-of-file bugs in the original server.)

5 Related Work

The performance problems of the CGI interface has led others to develop higher-speed alternatives [20, 25]. In fact, one of the driving motivations behind the Microsoft .NET initiative [17] appears to be the need to improve Web server performance, partially by eliminating process boundaries.[7] Apache provides a module interface that allows programmers to link code into the server that, among other things, generates content dynamically for given URLs. However, circumventing the underlying operating system's protection mechanisms without providing an alternative within the server opens the process for catastrophic failures.

FastCGI [20] provides a safer alternative by placing each content generator 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. It also complicates the creation of CGI protocols that communicate higher-order data, such as that of section 4.4.

IO-Lite [21] 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 MrEd 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 [25], in work on resource containers [6], 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 in MrEd provide similar functionality with a simpler API than those mentioned.

Like our server programs, Java servlets [8] 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 programmer with the task of manually marshaling state into and out of re-written urls, cookies, or secure socket layer connections.

The Java servlet interface allows implementations to distribute requests to multiple JVMs for automatic load balancing purposes. Lacking the subprocess management facilities of MrEd's custodians, they rely on an explicit delete method in the servlet to shut the subprocess down cooperatively. While this provides more flexibility by allowing arbitrary clean-up code, the servlet isn't guaranteed to comply.

Finally, the J-Server [24] runs atop Java extended with operating systems features. The J-Server team identified and addressed the server's need to prevent dynamic content generators from shutting down the entire server, while allowing the server to shutdown the content generators reliably. They too identified the

[7] Jim Miller, personal communication.

need for generators 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.

6 Conclusion and Future Work

The content of the Web is becoming more dynamic. The next generation of Web servers must therefore tightly integrate and support the construction of extensible and verifiable dynamic content generators. Furthermore, they must allow programmers to write interactive, dynamic scripts in a more natural fashion than engendered by current Web scripting facilities. Finally, the servers must themselves become more extensible and customizable.

Our paper demonstrates that all these programming problems can be solved with a high-level programming language, provided it offers OS-style services in a safe manner. Our server accommodates both kinds of extensibility found in traditional servers—applications, which serve data, and extensions, which adapt the behavior of the server itself—by exploiting its underlying module system. All these features are available on the wide variety of platforms that run MrEd (both traditional operating systems and experimental kernels such as the OS/Kit [12]). The result is a well-performing Web server that accommodates natural and flexible programming paradigms without burdening programmers with platform-specific facilities or complex, error-prone dynamic protocols. We have deployed this server for our book's widely accessed web site.

Two major areas of future work involve type systems and interoperability. Research should explore how the essential additions to Scheme—dynamically linkable modules that are first-class values, threads, custodians, and parameters—can be integrated in typed functional languages such as ML and how the type system can be exploited to provide even more safety guarantees. While MrEd already permits programmers to interoperate with C programs through a foreign-function interface, we are studying the addition of and interoperation between multiple safe languages in our operating system, so programmers can use the language of their choice and reuse existing applications [16].

Acknowledgments: We thank Matthew Flatt and Darren Sanders for their assistance.

References

1. Acme Labs. Web server comparisons. `http://www.acme.com/software/thttpd/benchmarks.html`.
2. Apache. `http://www.apache.org/`.
3. Arlitt, M. and C. Williamson. Web server workload characterization: the search for invariants. In *ACM SIGMETRICS*, 1996.

4. Aron, M., D. Sanders, P. Druschel and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Annual Usenix Technical Conference*, 2000. San Diego, CA.
5. Banga, G. and P. Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.
6. Banga, G., P. Druschel and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Third Symposium on Operating System Design and Implementation*, February 1999.
7. BrightPlanet. DeepWeb. `http://www.completeplanet.com/Tutorials/DeepWeb/`.
8. Coward, D. Java servlet specification version 2.3, October 2000. `http://java.sun.com/products/servlet/index.html`.
9. Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.
10. Flatt, M. and M. Felleisen. Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.
11. Flatt, M., R. B. Findler, S. Krishnamurthi and M. Felleisen. Programming languages as operating systems (*or*, Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, September 1999.
12. Ford, B., G. Back, G. Benson, J. Lepreau, A. Lin and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *16th ACM Symposium on Operating Systems Principles*, October 1997. Saint-Malo, France.
13. Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
14. Hughes, J. Generalising monads to arrows, 1998.
15. Kelsey, R., W. Clinger and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), October 1998.
16. Krishnamurthi, S. *Linguistic Reuse*. PhD thesis, Rice University, 2001.
17. Microsoft Corporation. `http://www.microsoft.com/net/`.
18. Mogul, J. The case for persistent connection HTTP. In *ACM SIGCOMM*, 1995.
19. NCSA. The common gateway interface. `http://hoohoo.ncsa.uiuc.edu/cgi/`.
20. Open Market, Inc. FastCGI specification. `http://www.fastcgi.com/`.
21. Pai, V. S., P. Druschel and W. Zwaenepoel. IO-lite: A unified I/O buffering and caching system. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.
22. Pitman, K. Special forms in lisp. In *Conference Record of the Lisp Conference*, August 1980. Stanford University.
23. Queinnec, C. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.
24. Spoonhower, D., G. Czajkowski, C. Hawblitzel, C.-C. Chang, D. Hu and T. von Eicken. Design and evaluation of an extensible web and telephony server based on the J-Kernel. Technical report, Department of Computer Science, Cornell University, 1998.
25. Thau, R. Design considerations for the Apache server API. In *Fifth International World Wide Web Conference*, May 1996.