

Hop, a Fast Server for the Diffuse Web

Manuel Serrano

Inria Sophia Antipolis, INRIA Sophia Antipolis 2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex, France

Abstract. The *diffuse* Web is an alternative way of using the Web 2.0 infrastructure for building personal diffuse applications. Systems that let users tune the temperature of their house with a cell-phone, check that the shutters are closed with a PDA, or select the music to be played on a Hi-Fi system with a PC are examples of the targeted applications. Diffuse Web applications have similarities with Web 2.0 applications: *i)* they rely on fast bi-directional interactions between servers and clients, and *ii)* they make extensive use of non-cachable dynamic contents. On the other hand, diffuse applications have also an important difference with respect to traditional Web applications: they generally do not need to deal with a huge number of simultaneous users. That is, diffuse Web applications are built on top of standard technologies but they use it differently. Therefore they demand different optimizations and tunings. Hop (<http://hop.inria.fr>) is a platform designed for building and running diffuse Web applications. Its software development kit contains two compilers, one interpreter, and a *bootstrapped* Web server. That is, the Hop Web server is implemented in Hop. This paper shows that this implementation strategy allows Hop to dramatically outperform the popular mainstream Web servers for delivering dynamic contents. Contrary to most servers, Hop delivers static and dynamic contents at a comparable pace. The paper details the implementation of the Hop Web server.

1 Introduction

The Web is the new ubiquitous platform where applications of tomorrow will be deployed. Though already wide, the Web will eventually become even wider when it connects all the appliances that surround us. The Web has already produced amazing new applications such as Google Map but a radically new way of thinking will be required.

Our answer to the challenge of programming ubiquitous Web applications relies on a small number of principles [31]. A Web application is not a federation of dynamic pages but a single, coherent program with multiple projections on servers or (perhaps roaming) clients. A typical application syndicates multiple data sources and, at the same time, is driven by multiple event streams.

Managing home appliances and organizing multimedia streams are typical targets for these new Web applications. Building these applications requires appropriate programming languages whose semantics, compilation process and

runtime environment must fit the technologies offered by the Web. This paper focuses on this latter aspect.

Hop is a platform for programming ubiquitous, or *diffuse*, Web applications [31,33,34]. Its development kit contains two compilers, one interpreter, and one Web server. The first compiler is in charge of generating the native code executed by the server-side of the application. The second compiler produces JavaScript code executed by the client-side. The interpreter, which resides on the server, is used for fast prototyping. It has poor speed performance but since it may call compiled code and vice versa without any overhead, its speed is generally not a performance bottleneck.

The Hop Web server has been specially designed and tuned for serving efficiently the HTTP requests needed by diffuse applications. This paper focuses on studying its performance. It will be shown that using a *bootstrapped* software architecture where the server is implemented and executed in the same runtime environment as that used to execute user programs, may lead to a major speed improvement in comparison to mainstream Web servers that rely on CGI or FastCGI protocols. More precisely, it will be shown that for serving dynamic contents, Hop outperforms traditional generalist Web servers by a factor that increases with the number of dynamic responses. The paper presents the software architecture and the implementation techniques deployed in Hop.

1.1 The context

The diffuse applications targeted by Hop use the Web differently than traditional Web 1.0 and 2.0 applications. They mostly use HTTP as a general purpose communication channel and they merely consider HTML and JavaScript as intermediate languages such as the ones used by compilers. The most important characteristics of diffuse Web applications are as follows.

- Most devices involved in the execution of a diffuse application may both act as server and as client. For instance, in a multimedia application, a PDA can be used to control the music being played as it can serve music files itself. That is, diffuse Web applications do not strictly implement a client-server architecture. They share some similarities with a peer-to-peer application.
- The applications frequently involve server-to-server, server-to-client, and client-to-server communications. For instance, a multimedia application playing music uses these communications for updating the state of all the GUIs controlling the music being played back.
- Programs are associated with URLs. Programs start when a Web browser requests such an URL. This generally loads a GUI on that browser. Apart from that initial transfer, most other communications will involve *dynamic contents* which can either be dynamic HTML documents or serialized data structures.
- The number of simultaneous concurrent requests is small because in general, only one user raises the temperature of the heating system or raises the volume of the Hi-Fi system. Hence dealing efficiently with a large number of connections is not a topmost priority for the servers considered here.

These characteristics have consequences on the implementation strategy a diffuse Web server should use. For instance, the first one, requires servers to have a small enough memory footprint to be embeddable inside small devices such as mobile phones. The second one requires servers to handle persistent connections efficiently. The third one demands short response times for dynamic documents. The fourth one impacts the concurrency model that should be deployed. Therefore, although diffuse Web applications probably have workloads that resemble those of Web 2.0 applications [25], they demand dedicated implementation strategies. In practice, general purpose Web servers that are mostly optimized for dealing with a large number of simultaneous connections are not suitable for most diffuse applications.

The vast majority of studies concerning the performance of Web servers concentrate on one crucial problem as old as the Web itself: sustaining the maximal throughput under heavy loads. This problem has been mostly addressed from a network/system programming perspective. For instance, a paper by Nahum *et al.* [26] surveys the impact on the performance of using various system services combinations. Another paper by Brech *et al.* explores and compares the different ways to accept new connections on a socket [6]. The seminal events-versus-threads dispute is more active than ever [1,40] and no clear consensus emerges. Hop is agnostic with respect to the concurrency model. Its software architecture supports various models that can be selected on demand.

The long debated question regarding kernel-space servers versus user-space servers seems to be now over. Initially it has been observed that kernel-space servers outperformed user-space servers [13], independently of the hosting operating systems. Adding zero-copy functions in the system API, such as the `sendfile`, has changed the conclusion [29]. In addition, another study [35] has shown that the gap between kernel-space and user-space that prevailed in older implementations unsurprisingly becomes less significant when the percentage of requests concerning dynamically generated content increases. Since Hop is designed mainly for serving dynamic contents, this study is of premium importance.

1.2 Organization of the paper

Section 2 presents the general Hop's software architecture. Then Section 3 shows how the architecture is actually implemented. Section 4 presents the overall performance evaluation of the Hop Web server and compares it to other Web servers. It shows that Hop is significantly faster than all mainstream Web servers for serving dynamic documents. Section 5 presents related work.

2 The implementation of the HOP Web server

This section presents the general implementation of the Hop Web server. It first briefly sketches the Hop programming language and its execution model. Then, it shows the general architecture of the server.

2.1 The HOP programming language and its implementation

Since the Hop programming language has already been presented in a previous paper [31], only its main features are exposed here. Hop shares many characteristics with JavaScript. It belongs to the functional languages family. It relies on a garbage collector for automatically reclaiming unused allocated memory. It checks types dynamically at runtime. It is fully polymorphic (i.e., the *universal* identity function can be implemented). Hop has also several differences with JavaScript, the most striking one being its parenthetical syntax closer to HTML than to C-like languages. Hop is a full-fledged programming language so it offers an extensive set of libraries. It advocates a CLOS-like object oriented programming [5]. Finally, it fosters a model where a Web application is conceived as a whole. For that, it relies on a single formalism that embraces simultaneously server-side and client-side of the applications. Both sides communicate by means of function calls and signal notifications. Each Hop program is automatically associated with a unique URL that is used to start the program on the server. The general execution pattern of a Hop program is as follows:

- When an URL is intercepted by a server for the first time, the server *automatically* loads the associated program and the libraries it depends on.
- Programs first authenticate the user they are to be executed on behalf of and they check the permissions of that user.
- In order to *load* or *install* the program on the client side, the server elaborates an abstract syntax tree (AST) and compiles it on the fly to generate a HTML document that is sent to the client.
- The server side of the execution can be either executed by natively compiled code or by the server-side interpreter. If performance matters compiled code has to be preferred. However, for most applications, interpreted code turns out to be fast enough.

Here is an example of a simple Hop program.

```
(define-service (hello)
  (<HTML> (<DIV> :onclick ~(alert "world!") "Hello")))
```

Provided a Hop server running on the local host, browsing the URL `http://localhost:8080/hop/hello` loads the program and executes the `hello` service. Contrary to HTML, Hop's markups (i.e., `<HTML>` and `<DIV>`) are node *constructors*. That is, the service `hello` *elaborates* an AST that is compiled on-the-fly into HTML when the result is transmitted to the client. It must be emphasized here, that a two phased evaluation process is strongly different from embedded scripting language such as PHP. The AST representing the GUI exists on the client as well as on the server. This brings flexibility because it gives the server opportunities to deploy optimized strategies for building and manipulating the ASTs. For instance, in order to avoid allocating a new AST each time a `hello` request is intercepted by the server, the AST can be elaborated at load-time and stored in a global variable that is reused for each reply.

```
(define hello-ast
  (<HTML> (<DIV> :onclick ~(alert "world!") "Hello")))

(define-service (hello) hello-ast)
```

2.2 The overall HOP Web server architecture

As most servers, when Hop intercepts a request it builds a reifying data structure which contains general informations such as the requested URL, the IP address of the client, and the date of the connection. In addition, Hop also proceeds to an early request authentication. That is, using the optional HTTP `authorization` field, it automatically searches in its database of declared users one whose login matches. If this query fails, a default *anonymous* user is associated with the request. This allows Hop to handle all requests on behalf of one particular user. Permissions to access the file system and to execute programs are granted individually, user by user.

Once users are authenticated and the request fully reified Hop builds a response object. This transformation is accomplished by a Hop program that can be changed as needed by users [32]. This gives flexibility to Hop that can therefore be used in various contexts such as smart proxies, application servers, load balancers, etc. It also strongly constraints its implementation because it forbids some classical optimizations. For instance it prevents Hop from re-using already allocated objects for representing requests because since these objects are used by user-defined programs they have dynamic extent.

Hop uses a traditional pipeline for processing HTTP requests, whose graphical representation is given in Figure 1. The advantages of using such an architecture for implementing Web servers have been deeply studied and are now well understood [42,44,9]. This Section presents the Hop pipeline without addressing yet the problem of scheduling its execution flow. In particular, it is not assumed here any sequential or parallel execution of the various stages. The scheduling strategies will be described in Sections 2.3 and 2.4.

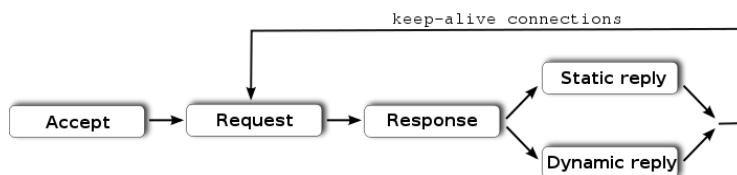


Fig. 1. The 4-stage HOP's pipeline.

In the Hop's pipeline, the first stage ("Accept"), establishes connections with clients. The second stage ("Request"), parses HTTP requests and builds the data

structure reifying the authenticated requests. The stage “Response” elaborates responses to the requests. As suggested by the work on SEDA [42], provision is taken to let the Hop scheduler handle static replies and dynamic replies differently. This is reflected by the pipeline fork after the “Response” stages. Persistent connections are handled by looping backward in the pipeline after the “Reply” stages. In order to avoid cluttering the next pipeline figures, keep-alive connections will be no longer presented.

The last two stages of the pipeline spawn user Hop programs executions. Services, such as `hello` defined in Section 2.1, are executed during the “Response” stage. For instance, when the server handles the request `http://localhost:8080/hop/hello` the “Response” stage establishes the connection between the absolute path of the request URL, namely `/hop/hello`, and the defined service `hello`. It then invokes the latter. In this particular case, this service returns an AST representing a HTML tree. The “Response” stage wraps the values returned by services into `response` objects that are used by the final “Reply” stages. The “Static reply” stage handles static files or constant strings. It simply sends the characters composing the response to the clients. The “Dynamic reply” stage handles all other cases. When a dynamic response is an AST, this stage traverses the structure for compiling it on the fly into a regular HTML or XHTML document. That traversal can be controlled by user programs. The AST is represented by an object hierarchy that can be extended by user programs and the methods that implement the traversal can be overridden.

Server side executions can either involve compiled codes or interpreted codes. Server-side interpreted code has not been optimized for performance but for flexibility. Hence the performance ratio between the two execution modes is strongly in favor of the former.

Flexibility is the main motivation for separating the elaboration of an AST and its compilation into a final document. As already mentioned in Section 2.1 this gives users the opportunity to *cache* ASTs. It also allows programs to reuse the same tree in different contexts. For instance, a same tree can be once compiled to HTML 4.01 and once to XHTML 1.0, depending on the capacities of the requesters that are identified in the HTTP request header.

2.3 HOP concurrency

Hop aims at alleviating as much as possible the complexity of programming diffuse applications on the Web. This motivation has deeply impacted the overall design of the language. For instance, the language relies on a garbage collector, higher order functions, full polymorphism, and transparent serialization for function calls that traverse the network, because all these features make programs easier to write and to maintain. Obviously, the concurrency model is also another fundamental aspect of the language which has been impacted by the general Hop’s philosophy. The concurrency model has been mainly designed with expressiveness and simplicity of programming in mind, more than runtime speed.

In Hop, responses to HTTP requests are all produced by user defined programs. This characteristic allows users to change the whole behavior of the server. This also deeply impacts its implementation because the concurrency model has to accommodate the spawning of user programs from various pipeline stages. Since running these programs may be unpredictably long, provisions have to be taken to execute them without blocking the entire server. That is, the server must still be able to answer other requests while executing user programs. This requires the server to be able to process multiple requests in parallel.

Although some previous studies might lead us to think that avoiding processes and threads by using an event-driven model can increase the speed [28,42], this form of concurrency forces programs to adopt a discipline that consists in splitting execution into a list of small call-backs that are scheduled by an event loop. Each of these call-backs must be as fast and short as possible in order to avoid monopolizing the CPU for too long. We have considered that organizing user programs into a well balanced list of call-backs was an unacceptable burden for the programmers. Hence, in spite of any performance considerations, we have decided to give up with pure event-driven models for Hop.

Currently Hop relies on a preemptive multi-threaded execution for user programs. However, the server and more precisely, the pipeline scheduler, is *independent* of the concurrency model of user programs, as long as they are executed in parallel with the stages of the pipeline. Hence, alternative concurrency models such as *cooperative threads* or *software memory transactions* could be used in Hop. This independence also allows many Web architectures to be used for scheduling the Hop pipeline. For instance, Hop may use multi-processes, multi-threads, pipeline [44,9], AMPED [28] or SYMPED [29], or a mix of all of them. In order to avoid early decisions, we have extracted the Hop scheduler from the core implementation of the server. That is, when the server is spawned, the administrator can select his scheduler amongst a set of predefined schedulers or provide his own. Hop pipeline schedulers are actually regular Hop user programs because the API that connects the scheduler to the server is part of the standard Hop library. The rest of this section emphasizes the simplicity of developing and prototyping new schedulers. In particular, it will be shown that adding a new scheduler is generally as simple as defining a new class and a couple of new methods. Using the server interpreter this can even be tested without any recompilation.

2.4 Hop pipeline scheduler

The Hop pipeline scheduler is implemented using a class hierarchy whose root is an abstract class named `scheduler`. Three methods `accept`, `spawn`, and `stage` implement the pipeline machinery. The Hop Web server provides several concrete subclasses of `scheduler` that can be selected using command line switches or by user programs. Extra user implementations can also be provided to Hop on startup. This feature might be used to easily test new scheduling strategies.

```
(abstract-class scheduler)

(define-generic (accept s::scheduler socket))
(define-generic (spawn s::scheduler proc . o))
(define-generic (stage s::scheduler th proc . o))
```

In this section, the general pipeline implementation is presented, followed by several concrete schedulers. For the sake of simplicity many details of the actual implementation are eluded. Exception handling is amongst them.

The pipeline implementation When the Hop Web server is started, it first creates a socket that listens to new connections. The function `make-server-socket` of the Hop standard API takes one mandatory argument, a port number, and one optional argument, a backlog size:

```
(define-parameter port 8080)
(define-parameter somaxconn 128)

(define socket-server (make-server-socket (port) :backlog (somaxconn)))
```

Once the command line is parsed and the *Runtime Command* file loaded, the pipeline scheduler is created by instantiating the class corresponding to the selected scheduler.

```
(define-parameter scheduling-strategy 'pool)
(define-parameter max-threads 20)

(define pipeline-scheduler
  (case (scheduling-strategy)
    ((nothread) (instantiate::nothread-scheduler))
    ((one-to-one) (instantiate::one-to-one-scheduler))
    ((pool) (instantiate::pool-scheduler (size (max-threads))))
    ...))
```

The main server loop is entered with:

```
(accept pipeline-scheduler socket-server)
```

The function `accept` is a *generic function* [5]. That is, a function whose default implementation might be overridden by *methods*. When a generic function is called, the actual implementation, i.e., the method to be executed, is chosen according to the dynamic types of the arguments of the call. The default implementation of `accept` is as follows:

```
1: (define-generic (accept S::scheduler serv)
2:   (let loop ()
3:     (let ((sock (socket-accept serv)))
4:       (spawn S stage-request sock 0)
5:       (loop))))
```

On line 3, a new connection is established. The request starts to be processed on line 4. The function `spawn` being also a generic function, its actual implementation depends on the dynamic type of the scheduler.

The `spawn` function requires at least two parameters: a scheduler (`S`) and a function (`stage-request`) which can be considered as a *continuation*. The function `spawn` starts an *engine* which calls the function with the scheduler `S`, the engine itself, and all the other optional parameters `spawn` has received. As it will be presented in the next sections, the concurrency model used for executing the pipeline entirely depends on the actual implementation of the scheduler which may override the definition of `spawn`. This gives the freedom to each scheduler to use an implementation of its own for *creating* and *spawning* new engines. That is, one scheduler may implement its engine with sequential functional calls, another one may implement it with threads, and a third one could implement it with processes.

The function `stage-request` implements the second stage of the pipeline. It parses the HTTP header and body in order to create the object denoting the HTTP request.

```
1: (define (stage-request S th sock tmt)
2:   (let ((req (http-parse-request sock tmt)))
3:     (stage S th stage-response req)))
```

The request is parsed on line 2. The function `http-parse-request` reads the characters available on the socket `sock` with a timeout `tmt`. A value of 0 means no timeout at all. Parsing the request may raise an error that will be caught by an exception handler associated with the running thread. This handler is in charge of aborting the pipeline. Once the *request* object is created and bound to the variable `req` (see on line 2), the third stage of the pipeline is entered. The function `stage` is the last generic function defined by the `scheduler` class. Although its semantics is equivalent to that of `spawn` there is a point in supporting two different functions. As it will be illustrated in the next sections, distinguishing `spawn` and `stage` is needed for enlarging the scope of possible scheduler implementations.

The function `stage-response` creates a *response* object from a *request* object. It is implemented as:

```
1: (define (stage-response S th req)
2:   (let* ((rep (request->response req))
3:         (p (if (http-response-static? rep)
4:                 stage-static-answer
5:                 stage-dynamic-answer))))
6:     (stage S th p req rep)))
```

The two functions `stage-static-answer` and `stage-dynamic-answer` being similar only one is presented here:

```
(define (stage-static-answer S th req rep)
  (stage-answer S th req rep))
```

Using two functions instead of one gives the scheduler the opportunity to deploy different strategies for dealing with static and dynamic requests [42].

```
(define (stage-answer S th id req rep)
  (let* ((sock (http-request-socket req))
        (conn (http-response rep sock)))
    (if (keep-alive? conn)
        (if (= (scheduler-load S) 100)
            ;; use a minimal timeout (1ms)
            (stage S th stage-request sock 1)
            ;; use the default timeout (3ms)
            (stage S th stage-request sock 3000))
        (socket-close sock))))
```

After this sketch of the pipeline implementation the next sections present several actual scheduler implementations.

The row pipeline scheduler Several Hop schedulers execute the stages of the pipeline sequentially, that is, they associate a new thread or a new process with each newly established connection that is used all along the pipeline. In order to alleviate the implementation of new schedulers that belong to this category, Hop provides a dedicated abstract class, namely `row-scheduler`, that overrides the `stage` generic function.

```
(abstract-class row-scheduler::scheduler)
```

```
(define-method (stage S::row-scheduler t p . o) (apply p S t o))
```

When no threads are used, jumping from one stage to another is implemented as a traditional function call. Hence, the implementation of the `stage` method of a `row-scheduler`, just consists in calling the function it has received as second argument.

The nothread pipeline scheduler The simplest form of scheduler implements no parallelism at all. Within an infinite loop, the `nothread` scheduler waits for a new connection to be established, it then executes in sequence all the stages of the pipeline and it loops back, waiting for new connections (see Figure 2).

Implementing the `nothread` scheduler is straightforward because it only requires to override the generic function `spawn` with a method that merely calls the procedure it receives with the optional arguments and a dummy thread that is created by the scheduler. This thread is never used but it is required for the sake of compatibility with the other schedulers.

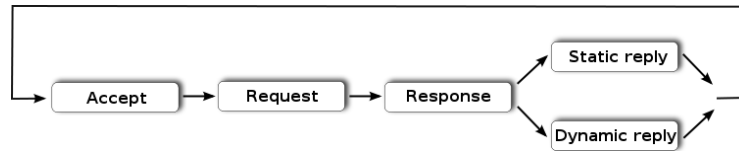


Fig. 2. The nothread pipeline scheduler.

```

(class nothread-scheduler::row-scheduler)

(define *dummy* #f)
(define-method (spawn S::nothread-scheduler p . o)
  (unless (thread? *dummy*) (set! *dummy* (instantiate::hopthread)))
  (apply stage S *dummy* p o))
  
```

The `nothread` scheduler is fast but unrealistic since it cannot handle more than one request at a time. Using such a scheduler would prevent Hop from being used for serving long lasting requests such as music broadcasting.

The one-to-one scheduler The `one-to-one` scheduler creates one new thread per connection (see Figure 3). Within an infinite loop it waits for connections. As soon as such a connection is established, it creates a new thread for processing the request. The main loop starts this new thread and waits for a new socket again.

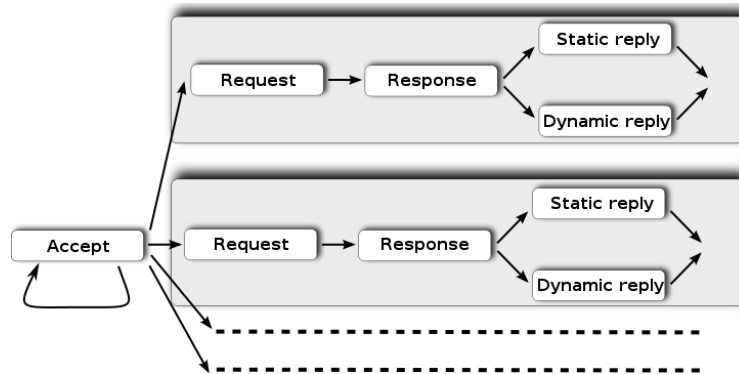


Fig. 3. The one-to-one pipeline scheduler.

Implementing the `one-to-one` scheduler is as simple as implementing the `nothread` scheduler. It only requires to override the `spawn` generic function.

```
(class one-to-one-scheduler::row-scheduler)

(define-method (spawn S::one-to-one-scheduler p . o)
  (letrec ((th (instantiate::hopthread
                (body (lambda () (apply p S th o))))))
    (thread-start! th)))
```

The `one-to-one` scheduler supports parallel execution for requests so it overcomes the major drawback of the `nothread` scheduler. It is easy to implement persistent HTTP connections using this scheduler because after a response is sent to the client, the same thread can check if new requests are pending on the socket. However, in spite of this progress, the `one-to-one` scheduler is still inefficient because the system operations involved in creating and freeing threads are expensive.

The thread-pool scheduler To eliminate the costs associated with the thread creation of the `one-to-one` scheduler, the `thread-pool` scheduler has been implemented. It is almost identical to the `one-to-one` scheduler with two noticeable differences: *i*) it uses a pool of early created threads, and *ii*) the *accept loop* is implemented inside each thread loop. That is, all the threads implement the same loop that executes all the stages of the pipeline (see Figure 4). Persistent connections are handled inside the same thread as the initial request. In scenarios where HTTP requests are sent to the server in sequence, this scheduler is able to avoid all context switches because a single thread executes the entire pipeline, from the “Accept” stage to the “Reply” stage. Context switches only occur when several requests are accepted in parallel.

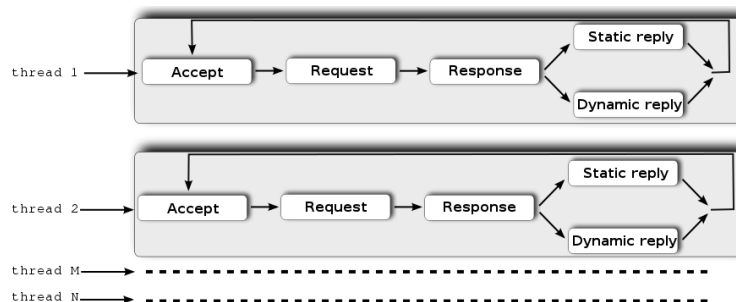


Fig. 4. *The thread-pool pipeline scheduler.*

The bookkeeping needed to manage the pool of threads makes the implementation of the `thread-pool` scheduler obviously more complex than the previous ones. As this is a classical problem of concurrent programming it is probably not

useful to present it here. Therefore all the details that are not strictly specific to Hop are therefore omitted.

The `pool-scheduler` class inherits from the `row-scheduler` class which it extends with two fields for holding the threads of the pool.

```
(class pool-scheduler::row-scheduler
  (threads::list read-only)
  (size::int read-only))
```

Each thread in the pool executes an infinite loop. When its action is completed a thread goes to `sleep` state. It will be awoken by the scheduler when a new connection will be assigned to this sleeping thread. Two functions implement this interface: `get-thread-pool` and `resume-thread-pool`.

```
(define-method (spawn S::pool-scheduler p . o)
  ;; get a free thread from the pool (may wait)
  (let ((th (get-thread-pool S)))
    (with-access::hopthread th (proc)
      ;; assign the new task to the thread
      (set! proc (lambda (S t) (stage S t p o)))
      ;; awake the sleeping thread
      (resume-thread-pool! th)
      th)))
```

Contrary to other schedulers, the call to `socket-accept` that waits for new connections is not invoked from the server main loop but inside each thread started by the scheduler. This is implemented by overriding the generic function `accept` for the `pool-scheduler` class and by creating an new function for implementing the “Accept” stage.

```
(define-method (accept S::pool-scheduler serv)
  (for (i 0 (pool-scheduler-size S))
    (spawn S stage-accept)))

(define (stage-accept S th)
  (let loop ()
    (let ((sock (socket-accept serv)))
      (stage S th stage-request sock 0)
      (loop))))
```

Other schedulers Other schedulers have been implemented inside Hop. In particular we have tried a scheduler inspired by the `cohort` scheduling [16] (see Figure 5), a scheduler using an `accept-many` strategy [6], and a scheduler using a queue of waiting tasks. Early observations yield us to think that none performs faster than the `thread-pool` scheduler for our targeted application field.

The cohort scheduling experienced in Hop consists in grouping threads by tasks rather than by requests. It is hard to implement and even harder to op-

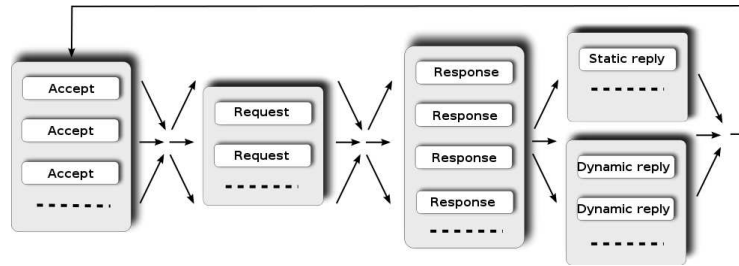


Fig. 5. The cohort pipeline scheduler.

optimize so up to now we have not been able to achieve good performance with it.

The *queue* strategy consists in associating stages of the pipeline to tasks. When a task must be executed, one thread is extracted from the pool. When the task completes, the thread goes back to the pool. A straightforward optimization of this scheduler removes superfluous queue operations and allows this scheduler to handle request in a row. When a thread should go back to the queue, it first checks if its queue of available threads is empty or not. If not empty, the same thread is used to execute the next stage of the pipeline.

The *accept-many* strategy consists in modifying the `accept` stage of the `thread-pool` scheduler in order to accept many connections at a time for purging as quickly as possible the socket backlog. All the connections that are established at a time are then processed by a pool of threads as the `thread-pool` scheduler does. Although the authors of the `accept-many` technique report significant speed acceleration when deploying this strategy in mainstream Web servers, it fails at accelerating Hop. The reason for this different behavior is probably to be searched in the application field targeted by Hop where massively parallel requests burst are rare.

3 Optimizations

The online paper *Server Design* [10] highlights three major reasons for a Web server to behave inefficiently: *i*) data copies, *ii*) memory allocations, and *iii*) context switches. According to this paper, an ideal server would be a server that avoids them all. Of course, this is not practically feasible but still, working as hard as possible on these issues improves the chances of success in the quest of a fast server. Section 2.4 has shown that some Hop schedulers are able to *accept*, *parse*, and *reply* to requests without any context switch. Section 2.4 has presented an example of such a scheduler. This section shows how Hop addresses the two other points.

3.1 Limiting the memory allocation

High level programming languages such as Hop help programmers by supporting constructions that abstract low level mechanisms involved at runtime. Providing high level powerful forms alleviates programmers from tedious tasks but it also generally comes with a price: it makes writing efficient programs more difficult. A major reason for this inefficiency is excessive memory allocations.

Excessive memory allocation dramatically limits the performance for two main reasons: *i)* programs spend a significant percentage of their execution to allocate and free memory chunks and, *ii)* it introduces additional context switches for parallel executions that run in shared-memory environments. When the heap is a shared resource, allocating and freeing memory require some sort of synchronization. In general this is implemented with locks that eventually conduct to context switches. Hence, polishing a thread scheduling strategy can be pointless if, at the same time, memory allocation is not tamed.

Version 1.8.7 of Hop allocates 47.5MB for serving 10,000 times a file of 512 bytes. The same test run on the version 1.10.0-pre3 allocates only 4.3MB, that is 457 bytes per request. This section presents the two transformations that have lead us to shrink memory by more than 10 times.

BglMem Contrary to a popular belief, garbage collectors generally used in high level languages impose no or minor runtime overhead [45]. The inefficiency of high level languages is more to be searched in the *implicit* memory allocations that are potentially hidden everywhere in the programs. For instance, calling a function might allocate lists if it accepts a variable number of arguments, declaring a function might yield to creating a closure if it makes use of free variables, opening a file or a socket might allocate inadequate buffers, etc.

In order to help programmers tame memory allocations, the Hop development kit provides an exact memory profiler. A dynamic tool keeps trace of the exact call graph at each allocation points. An offline tool produces histograms that, for each function of the programs, show: *i)* the number of data structures that have been allocated *by this function* and the overall size in bytes these allocations represent, and *ii)* the number of data structures and the size in bytes for all the functions the function dynamically calls. Using BglMem, we have, for instance, easily discovered that during a Hop test consisting in replying to 10,000 requests, the Hop function `http-parse-method-request` has allocated 60,006 pairs for a total of 468KB and 30,003 strings for a total of 654KB. In addition, the histograms produced by BglMem show that one of the children of this function has allocated 10,000 data structures representing the HTTP requests for a total size of 976KB.

The next sessions shows how BglMem has been used to reduce the Hop allocation and memory footprint.

One transformation Because of the size constraint of the paper, it is not possible to present the exhaustive list of optimizations that have been applied to

Hop to reduce its memory allocation. Hence only one exemplar transformations is presented here. It gets rid of implicit memory allocations that, once spotted are straightforward to eliminate.

Optimizing IO buffers: The function `socket-accept` waits for new connections on a socket. Once established the connection is associated with an input port for reading incoming characters and an output port for writing outgoing characters. BglMem reported that with Hop 1.8.7, `socket-accept` was responsible for allocating about 10MB of strings of characters! These 10MB came from the default configuration of `socket-accept` that creates a buffer of 1024 bytes for the input port associated with a connection. Removing these buffers has saved $10000 * 1024$ bytes of memory.

Hop uses sockets in a way that allows it to use its own memory management algorithm which is more efficient than any general purpose memory manager. Hop needs exactly as many buffers as there are simultaneous threads parsing HTTP requests. Hence, the obvious idea is to associate one buffer per thread not by socket. This can be implemented more or less efficiently depending on the nature of the scheduler. Some schedulers such as the `nothread-scheduler` or `pool-scheduler` (see Section 2.4 and Section 2.4) accept an optimal solution that totally eliminates the need to allocate buffers when the server is ready to accept requests. It consists in allocating one buffer per thread when the scheduler is initialized. These buffers are then simply reset each time the thread they are associated with is about to parse characters. The modification in the source code is minor. It only requires one additional line of code:

```
(define-method (accept S::nothread-scheduler serv)
  ;; create a buffer before the thread loop
  (let ((buf (make-string 1024)))
    (let loop ()
      ;; reuse buffer for each connection
      (let ((sock (socket-accept serv :inbuf buf)))
        (spawn S stage-request sock 0)
        (loop))))))
```

Wrap up Memory allocation has been measured on a test that consists in serving 10,000 a 512 bytes long file, without persistent connection. Each HTTP request of this memory test contained 93 bytes. Parsing each request produces a data structure made of strings (for denoting the path of the request, the client hostname, etc.), lists (for holding the optional lines of the HTTP header), symbols (for the HTTP version, the transfer encoding, the mime type, etc.) and an instance of the class `http-request` for packaging the parsed values plus some extra values such as a time stamp, an authenticated user, etc. In the test this structure uses the whole 457 bytes allocated per request. That is, applying the optimizations described in this section has successfully removed *all* the memory allocations not strictly needed for representing the HTTP requests. In particular, all the hidden allocations that can take place in high level languages such as

Hop have been eliminated. The current version of Hop is then close to optimal regarding memory allocation.

3.2 Persistent connections

Persistent connections have been one of the main reasons of the creation of HTTP/1.1. Two early papers report that significant accelerations can be expected from implementing persistent connections [21,27]. A client has two means for discovering that it has received the full body of a response: *i*) the connection is closed by the server or, *ii*) the length of the response to be read is explicitly provided by the server. Persistent connections, of course, can only use the second method.

Providing the length of a static response, i.e, a file or a string of characters, is straightforward (although some Web servers, such as Lighttpd, implement caches to eliminate repetitive calls to the expensive `fstat` system operation). Providing the size of a dynamic response is more challenging. One solution consists in writing the response in an auxiliary buffer first, then getting the length of that buffer and then writing the length and the buffer to the socket holding the connection. This technique is generally inefficient because it is likely to entail data copies. The characters have to be written to the buffer first which might need to be expanded if the response is large. Then, this buffer has to be flushed out to the socket, which is also likely to use a buffer of its own.

Hop uses a solution that avoids auxiliary buffers. It relies on *chunked* HTTP responses that break the response body in a number of chunks, each prefixed with its size. Using chunked responses in Hop is possible because of a dedicated facility provided by its runtime system. The Hop I/O system allows programs to associate *flush hooks* with input and output ports. Output flush hooks are functions accepting two arguments: the port that is to be flushed and the number of characters that are to be flushed out. The values produced by calling these hooks are directly written out to the physical file associated with the port before the characters to be flushed out.

Using output flush hook, chunked responses can be implemented as:

```
(define (chunked-flush-hook port size)
  (format "\r\n-x\r\n" size))
```

Using output port flush hooks is efficient because it imposes no overhead to the I/O system. The written characters are stored in a buffer associated with the output port, *as usual*. When the buffer is flushed out, the chunk size is written by the hook. Writing the chunk size is the only extra operation that has been added to answering responses. It is thus extremely lightweight and it allows persistent connections to be implemented efficiently (i.e., without extra copy nor extra memory allocation) for dynamic documents as well. Chunked HTTP responses have probably been designed for enabling this kinds of optimization but we have found no trace of equivalent techniques in the literature.

3.3 Operating system interface

Implementing fast I/Os with sockets requires some operating system tunings and optimizations. This section presents two of them.

Non-copying output Several studies have measured the benefit to be expected from using *non-copying* output functions such as the Linux `sendfile` facility [26,29]. This system service is supported by Hop. In addition to be fast because it avoids data copies and user-space/kernel-space traversals, it also simplifies the implementation of servers because it makes memory caches useless for serving static files. As in a previous study [29], we have noticed no acceleration when using memory cache for serving files instead of using a pure `sendfile`-based implementation. Using `sendfile` or a similar function actually *delegates* the caching to the operating system which is likely to perform more efficiently than a user-land application.

Network programming The default setting of sockets uses the Nagle algorithm [24] that is aimed at avoiding TCP packets congestion. We know from previous studies that this algorithm combined with the acknowledgment strategy used by TCP may cause an extra 200ms or 500ms delay for delivering the last packet of a connection [12]. This is called the *OF+SFS* effect that has been identified to be due to the *buffer tearing* problem [22]. Persistent HTTP connections are particularly keen to exhibit this problem and thus it is recommended to disable the Nagle algorithm for implementing more efficiently Web servers that support persistent HTTP connections [27]. Therefore, Hop supports configuration flags that can enable or disable the Nagle algorithm.

The `TCP_CORK` hybrid solution (or a *super-Nagle* algorithm) is supported by some operating systems. However, as reported by Mogul & Mingall [22] this socket option does not solve the OF+SFS problem in presence of HTTP persistent connections. We confirm this result because in spite of several attempts we have failed to eliminate the 200ms delay it sometimes imposes (in between 8 and 20% of the responses according to Mogul & Mingall, much more according to our tests) between two persistent connections. Hence, we gave up on using it.

Other configurations impact the overall performance of socket based applications. For instance, previous studies have suggested that an adequate calibration on the backlog of a listen socket (controlled by the system limit `somaxconn`) may improve significantly the performance [4]. For the workloads used to test Hop we have found that mid-range values (such as 128) yield better results.

4 Performance study

Although most HTTP requests involved in diffuse applications address dynamic contents, they also use static file transfers for cascading style sheets, images, and client-side libraries. Hence a fast server for the diffuse Web should be able to deliver efficiently dynamic *and* static documents. In this section we compare Hop to other Web servers for serving the two kinds of requests.

4.1 Performance evaluation caveat

This paper focuses on the performance evaluation of the Hop Web server, which is not to be confused with a performance evaluation of the Hop server-side programming language. Hop relies on the Bigloo compiler for generating server-side native code. It has already been shown that this compiler delivers native code whose performance is only 1.5 to 2 times slower than corresponding C code [30]. That is, the Hop server-side compiled code significantly outperforms the popular scripting languages such as PHP, Ruby, Python, as well as bytecode interpreted languages such as Java. In order to avoid overemphasizing the performance of the Hop programming language against PHP, Ruby, Java, or even C, we have only used simplistic generated documents that require minimalist computations. Our typical generated documents only require a few “prints” to be produced. Restricting to simple documents minimizes the performance penalty imposed by slow scripting languages implementations and allows us to focus on the evaluation of the mechanisms used by the server for running user programs on the server-side of the application.

4.2 Experimental Environment

Our experimental environment consists of three computers connected to a LAN: *i)* a server made of a bi-processor Intel PIV Xeon, 3Ghz with 1GB of RAM running Linux-2.6.27, *ii)* a first client running Linux-2.6.27 executed by an Intel Core 2 Duo ULV 1.06Ghz with 1.5GB of RAM, and *iii)* a second client running Linux-2.6.27 executed by an Intel Code 2 Duo 2.0Ghz with 2GB of RAM. The network traffic is ensured by a Cisco Gigabit ethernet router Catalyst 3750G. Using the Unix tool `iperf` we have experimentally verified that this setting indeed permits ethernet frames to be transferred at the pace of one gigabit per second.

In this paper, the workloads used for our tests are generated by `httperf` [23] version 0.9.0, a tool dedicated to measuring the performance of Web servers.

Before concentrating on the actual performance of the servers, we have measured the requests rate our setting can sustain. Following the protocol suggested by Titchkosky *et al* [38], we have observed that our clients can sustain a combined workload of more than 6,000 requests per second, which is enough to saturate the tested servers.

4.3 Hop vs other Web servers

There are so many Web servers in vicinity that it is impossible to test them all¹. Hence, we have tried to select a representative subset of existing servers. We have used two mainstream servers implemented in C, one mainstream server implemented in Java, and two servers implemented in functional languages:

¹ At the time this paper has been written, the wikipedia articles comparing Web servers described 68 general purpose servers and 79 lightweight Web servers! See http://en.wikipedia.org/wiki/Comparison_of_web_servers.

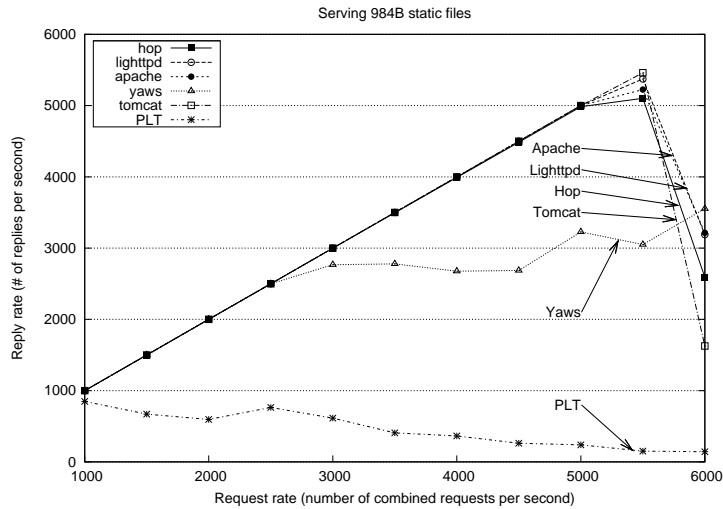


Fig. 6. Server performance. These tests measure the throughput a web server can sustain when delivering 984 bytes long static files. Each session consists in 5 consecutive requests that are sent using a single persistent connection.

- **Apache-2.2.10**, a popular Web server implemented in C. For producing dynamic documents with Apache, we have measured the performance of **mod_perl-2.0.4** and **mod_php5**, which both rely on the FastCGI protocol.
- **Lighttpd-1.4.20**, another popular Web server implemented in C. It is frequently used as a replacement of Apache on embedded devices such as routers and NASes.
- **Tomcat-5.5.27**, the popular Web server implemented in Java that relies on JSP for producing dynamic documents.
- **Yaws-1.77**, a small server implemented in Erlang [3].
- **PLT Web server-4.1.2**, a web server implemented in PLT-Scheme [14,15,43].

For this experiment all servers are used with their default configuration except for logging that has been disabled when possible. The Hop default configuration is as follows: *i*) use the `thread-pool` pipeline scheduler with 20 threads, *ii*) `somaxconn` = 128, *iii*) initial heap size = 4MB, *iv*) keep-alive timeout = 5 seconds, *v*) the socket send buffer size is 12KB (as recommended by [27]).

As much as possible we have tried to write comparable versions of the dynamic test. That is for each of the tested languages, namely PHP, Perl, JSP, Erlang, Scheme, and Hop, we have tried to write a *fair* version of the test, that is a version as equivalent as possible to the version of the other languages.

Figures 6 and 7 presents the performance evaluation which calls for several observations:

Observation 1: In the considered application context where only a small number of users simultaneously connect to the server, Hop is one of the fastest

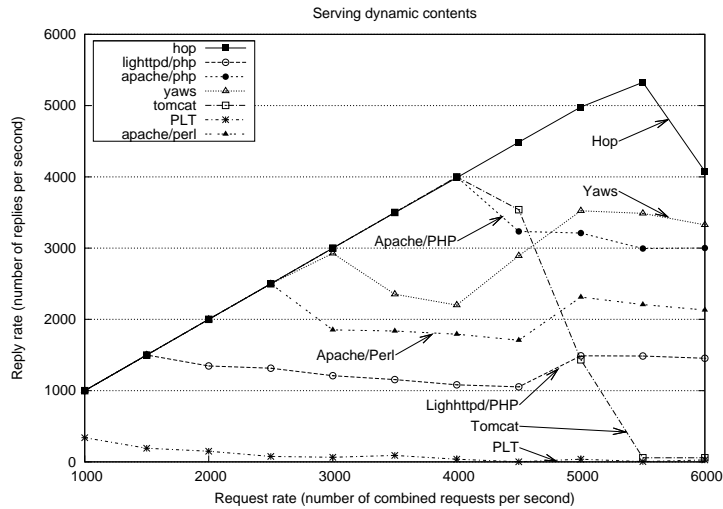


Fig. 7. *Server performance.* These tests measure the throughput a web server can sustain when delivering dynamic contents. Each session consists in 5 consecutive requests that are sent using a single persistent connection.

server for delivering static content. In particular, it is as fast as C servers such as Apache and Lighttpd. The test presented in the paper involves serving a file of about 1KB. We have conducted a second test that involves a file of 64KB. This second test confirms the results presented in Figure 6. The speed hierarchy of Web servers for serving a 64KB file is roughly the same as the one found when serving the 1KB file.

Observation 2: Hop is the fastest server for delivering dynamic content.

Observation 3: Hop and Yaws are the only two servers that deliver static content and dynamic content at approximately the same speed.

Observation 4: Yaws runs remarkably steady. In particular, its performance seems hardly impacted when the server load increases. Further investigation would be needed but this early result seems to demonstrate that the advantage claimed by the Erlang programming language [2] for dealing with massively concurrent applications is also observable for Web servers. Those concerned by overloaded servers should probably consider using message passing as fostered by Erlang.

Observation 5: PHP and Perl present comparable performance. They are both significantly slower than Hop.

Tomcat performance drops dramatically around 4000 requests per second because its memory footprint becomes too large (more than 700MB), which forces the operating system to start swapping the main memory. It should be noticed here that none of the servers is stopped and restarted when the load increases. Hence if a Web server leaks memory its performance will continuously

slow down. The PLT Scheme server suffers from the same problem as Tomcat. Its memory footprint rapidly approaches 1GB of memory. Hence, very soon it is not able to do anything useful because it is swapped out. Tomcat and PLT show how important it is to restrain memory allocation. An excessive memory allocation dramatically reduces the performance of a server.

From these observations, we can draw some conclusions. The experiment emphasizes persistent connections even for dynamic content delivery because each client emits consecutively 5 requests using the same connection. This corresponds to a real use-case for the targeted applications but it strongly penalizes all the systems that are not able to implement them efficiently. This might explain the poor performance of Apache and Lighttpd servers.

The file 984 bytes long file used for testing static delivery is exactly the file that gets generated by the dynamic content test. Hence, the static delivery and the dynamic delivery end up with the same characters written to the socket. The only difference between the two tests is the way these characters are obtained: from a static file in the first case, from an in-memory computation in the second. Hop and Yaws are the two fastest servers for delivering dynamic contents. They are both bootstrapped. This might be considered as an indication that bootstrapped Web servers should outperform other servers for delivering dynamic content. Tomcat is bootstrapped too but since its performance is deeply impacted by its excessively large memory footprint, no conclusion can be drawn from studying its performance.

5 Related work

Many high level languages have been used to implement Web servers but actually only a few of them can be compared to Hop. Functional languages have a long standing tradition of implementing Web servers, probably pioneered by Common Lisp that, as early as 1994, was already concerned by generating dynamic HTML content [18]! Today, Haskell seems very active with HWS [19], Wash/CGI [37], and HSP [7]. HWS is a server that uses a four-stages pipeline and a `one-to-one` scheduler. It relies on user-threads instead of system-threads for handling each requests. User-threads work well as long as no user program can be spawn from the stages of the pipeline. This probably explains why HWS is not able to serve dynamic content. HSP is a Haskell framework for writing CGI scripts. It used to be implemented as an Apache module [20] and but it is now hosted by a dedicated server HSP(r) based on HWS. Unfortunately HSP(r), as well as Wash/CGI, is incompatible with the currently released version of GHC, the Haskell compiler. Hence we have not been able to test any of them.

Smalltalk has Seaside which is one of the precursors in using continuations for modeling Web interactions [11]. It would have been interesting to measure its performance because Smalltalk, as Hop is a dynamic programming language but unfortunately it was not possible to install it on our Linux setting.

The impact of the concurrency model on the performance of the servers has been largely studied and debated but no clear consensus prevails yet. Some pre-

tend that an event-based model is superior to a thread model. Some pretend the contrary. A third group claims that blending the two models should be preferred [41]! Hence, the idea of proposing system independent models of the concurrency has emerged. Jaws is an adaptive Web server framework that allows programmers to implement their own Web server using on-the-shelf components [36]. This framework provides elementary blocks that can manage a pipeline, normalize URLs, support various styles of I/Os, etc. Combining these components simplifies the development of a server without penalizing its performance.

Flux [8] and Aspen [39] are two programming languages that allow programmers to choose, at compile-time, the concurrent model used at run-time. Flux consists in a set of syntactic extensions to C/C++ that are expanded, at compile-time, into regular C/C++ programs. Aspen like Erlang [3], eschews shared memory in favor of message passing. The parallel structure of an Aspen program is specified independently of its computational logic and, at run-time, Aspen dynamically allocates threads according to the dynamic workload of the server. Prototypical Web servers have been implemented in Flux and Aspen that show performance not significantly better than Apache for static and dynamic contents.

Saburo is an Aspect Oriented framework for generating concurrent programs [17]. The PhD thesis introducing Saburo focuses on the performance of Web servers. Contrary to Hop that relies on a dynamic selection of the concurrency model (implemented by means of classes and late binding), Saburo as Flux and Aspen relies on a static model. In theory these systems should then be able to perform faster than Hop because they have opportunities to optimize the implementation of the concurrency model at compile-time. In practice, the Hop implementation of the late binding used in the pipeline scheduler is fast enough not to impact the overall performance of the server.

6 Conclusion

This paper presents the Hop server that is mainly designed for running diffuse applications on the Web. The paper presents its versatile architecture that supports various concurrent programming models as well as significant parts of its implementation.

The paper shows that programming an efficient server for the diffuse Web is not only a problem of good system and network practices, although these still have a large impact on the overall performance. The new problem is to *combine*, fast network and system programming *and* fast interactions between the server main loop that deals with HTTP requests and the user helper programs that produce responses.

The solution supported by Hop consists in merging, inside a single runtime environment, the server main loop and the user programs. This can be build by using the same programming language for implementing the server itself and the user programs. Such a *bootstrapped* Web server can eliminate all communication costs between the server main loop and user programs. Hence, it can outperform

traditional general purpose Web servers that handle user programs as external processes. The Hop Web server that delivers dynamic documents significantly faster than all the other tested servers shows this can be achieved using high level dynamic programming languages.

References

- [1] Adya, A. *et al.* – **Cooperative Task Management without Manual Stack Management or Event-driven Programming is Not the Opposite of Threaded Programming** – Proceedings of the Usenix Annual Technical Conference, Monterey, CA, USA, Jun, 2002, pp. 289–302.
- [2] Armstrong, J. *et al.* – **Concurrent Programming in ERLANG** – Prentice Hall, 1996.
- [3] Armstrong, J. – **Concurrency Oriented Programming in Erlang** – Invited talk of the FFG conference, 2003.
- [4] Banga, G. and Druschel, P. – **Measuring the Capacity of a Web Server** – USENIX Symposium on Internet Technologies and Systems, 1997.
- [5] Bobrow, D. *et al.* – **Common lisp object system specification** – special issue, Notices, (23), Sep, 1988.
- [6] Brech, T. and Pariag, D. and Gammo, L. – **accept()able Strategies for Improving Web Server Performance** – Proceedings of the USENIX 2004 Annual Technical Conference, Boston, MA, USA, Jun, 2004.
- [7] Broberg, N. – **Haskell Server Pages through Dynamic Loading** – Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, Tallinn, Estonia, 2005, pp. 39–48.
- [8] Burns, B. *et al.* – **Flux: A Language for Programming High-Performance Servers** – In Proceedings of USENIX Annual Technical Conference, 2006, pp. 129–142.
- [9] Choi, G. S. *et al.* – **A Multi-Threaded PIPELINED Web Server Architecture for SMP/SoC Machines** – WWW '05: Proceedings of the 14th international conference on World Wide Web, Chiba, Japan, 2005, pp. 730–739.
- [10] Darcy, J. – **Server Design** – <http://pl.atyp.us/content/tech/servers.html>, Aug, 2002.
- [11] Ducasse, S. and Lienhard, A. and Renggli, L. – **Seaside - a multiple control flow web application framework** – Proceedings of the ESUG Research Track, 2004.
- [12] Heidemann, J. – **Performance Interactions Between P-HTTP and TCP Implementations** – ACM Computer Communication Review, 27(2), April, 1997, pp. 65–73.
- [13] Joubert, P. *et al.* – **High-Performance Memory-Based Web Servers: Kernel and User-Space Performance** – Usenix, 2001, pp. 175–188.
- [14] Krishnamurthi, S. – **The CONTINUE Server (or, How I Administrated PADL 2002 and 2003)**. – Practical Aspects of Declarative Languages, New Orleans, LA, USA, Jan, 2003, pp. 2–16.
- [15] Krishnamurthi, S. *et al.* – **Implementation and Use of the PLT Scheme Web Server – Higher Order and Symbolic Computation**, 20(4), 2007, pp. 431–460.
- [16] Larus, J. and Parkes, M. – **Using Cohort Scheduling to Enhance Server Performance** – Proceedings of the Usenix Annual Technical Conference, Monterey, CA, USA, Jun, 2002, pp. 103–114.
- [17] Loyaute, G. – **Un modèle génératif pour le développement de serveurs internet** – Université Paris-Est, Paris, France, Sep, 2008.
- [18] Mallery, J. C. – **A Common LISP Hypermedia Server** – In Proc. First International World-Wide Web Conference, 1994, pp. 239–247.
- [19] Marlow, S. – **Writing High-Performance Server Applications in Haskell, Case Study: A Haskell Web Server** – Haskell '00: Proceedings of the ACM SIGPLAN Haskell Workshop, Montreal, Canada, Sep, 2000.
- [20] Meijer, E. and Van Velzen, D. – **Haskell Server Pages – Functional Programming and the Battle for the Middle Tier Abstract** – Haskell '00: Proceedings of the ACM SIGPLAN Haskell Workshop, Montreal, Canada, Sep, 2000.
- [21] Mogul, J. C. – **The case for persistent-connection HTTP** – SIGCOMM '95: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication, Cambridge, Massachusetts, United States, 1995, pp. 299–313.
- [22] Mogul, J. C. and Minshall, G. – **Rethinking the TCP Nagle algorithm** – SIGCOMM Comput. Commun. Rev., 31(1), New York, NY, USA, 2001, pp. 6–20.
- [23] Mosberger, D. and Jin, T. – **httperf: A tool for Measuring Web Server Performance** – In First Workshop on Internet Server Performance, 1998, pp. 59–67.
- [24] Nagle, J. – **Congestion Control in IP/TCP Internetworks** – RFC 896, Internet Engineering Task Force, Jan, 1984.

- [25] Nagpurkar, P. *et al.* – **Workload Characterization of selected JEE-based Web 2.0 Applications** – Proceedings of the IISWC 2008. IEEE International Symposium on Workload Characterization, Sep, 2008, pp. 109–118.
- [26] Nahum, E. and Barzilai, T. and Kandlur, D. D. – **Performance Issues in WWW Servers** – IEEE/ACM Transactions on Networking, 10(1), Feb, 2002.
- [27] Nielsen, H. F. *et al.* – **Network Performance Effects of HTTP/1.1, CSS1, and PNG** – Proceedings of the ACM SIGCOMM'97 conference, Cannes, France, Sep, 1997.
- [28] Pai, V. S. and Druschel, P. and Zwaenepoel, W. – **Flash: An efficient and portable Web server** – Proceedings of the Usenix Annual Technical Conference, Monterey, CA, USA, Jun, 1999.
- [29] Pariag, D. *et al.* – **Comparing the Performance of Web Server Architectures** – SIGOPS Oper. Syst. Rev., 41(3), New York, NY, USA, 2007, pp. 231–243.
- [30] Serpette, B. and Serrano, M. – **Compiling Scheme to JVM bytecode: a performance study** – 7th Sigplan Int'l Conference on Functional Programming (ICFP), Pittsburgh, Pennsylvania, USA, Oct, 2002.
- [31] Serrano, M. and Gallesio, E. and Loitsch, F. – **HOP, a language for programming the Web 2.0** – Proceedings of the First Dynamic Languages Symposium, Portland, Oregon, USA, Oct, 2006.
- [32] Serrano, M. – **The HOP Development Kit** – Invited paper of the Seventh ACM sigplan Workshop on Scheme and Functional Programming, Portland, Oregon, USA, Sep, 2006.
- [33] Serrano, M. – **Programming Web Multimedia Applications with Hop** – Proceedings of the ACM Sigmam and ACM Siggraph conference on Multimedia, Best Open Source Software, Augsburg, Germany, Sep, 2007.
- [34] Serrano, M. – **Anatomy of a Ubiquitous Media Center** – Proceedings of the Sixteenth Annual Multimedia Computing and Networking (MMCN'09), San Jose, CA, USA, Jan, 2009.
- [35] Shukla, A. *et al.* – **Evaluating the Performance of User-Space and Kernel-Space Web Servers** – CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, Markham, Ontario, Canada, 2004, pp. 189–201.
- [36] Smith, D. C. and Hu, J. C. – **Developing Flexible and High-performance Web Servers with Frameworks and Patterns** – ACM Computing Surveys, 301998.
- [37] Thiemann, P. – **WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms** – Practical Aspects of Declarative Languages, 2002.
- [38] Titchkosky, L. and Arlitt, M. and Williamson, C. – **A performance comparison of dynamic Web technologies** – SIGMETRICS Perform. Eval. Rev., 31(3), New York, NY, USA, Dec, 2003, pp. 2–11.
- [39] Upadhyaya, G. and Pai, V. S. and Midkiff, S. P. – **Expressing and Exploiting Concurrency in Networked Applications with Aspen** – PPOPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, San Jose, California, USA, 2007, pp. 13–23.
- [40] Von Behren, R. and Condit, J. and Brewer, E. – **Why Events Are A Bad Idea (for higher-concurrency servers)** – Proc. of HotOSIX: the 9th Workshop on Hop Topics in Operating Systems, Lihue, Hawaii, USA, May, 2003.
- [41] Welsh, M. *et al.* – **A Design Framework for Highly Concurrent Systems** – Berkeley, CA, USA, 2000.
- [42] Welsh, M. and Culler, D. and Brewer, E. – **SEDA: An Architecture for Well-Conditioned, Scalable Internet Services** – Symposium on Operating Systems Principles, 2001, pp. 230–243.
- [43] Welsh, N. and Gurnell, D. – **Experience report: Scheme in commercial Web application development** – ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, Freiburg, Germany, 2007, pp. 153–156.
- [44] Yao, N. and Zheng, M. and Ju, J. – **Pipeline: A New Architecture of High Performance Servers** – SIGOPS Oper. Syst. Rev., 36(4), New York, NY, USA, 2002, pp. 55–64.
- [45] Zorn, B. – **The Measured Cost of Conservative Garbage Collection** – Software — Practice and Experience, 23(7), Jul, 1993, pp. 733–756.