Writing High-Performance Server Applications in Haskell
Case Study: A Haskell Web Server

Simon Marlow
Microsoft Research Ltd., Cambridge
simonmar@microsoft.com

14th August 2000

Abstract

Server applications, and in particular network-based server applications, place a unique combination of demands on a programming language: lightweight concurrency and high I/O throughput are both important.

This paper describes a prototype web server written in Concurrent Haskell, and presents two useful results: firstly, a conforming server could be written with minimal effort, leading to an implementation in less than 1500 lines of code, and secondly the naive implementation produced reasonable performance. Furthermore, making minor modifications to a few time-critical components improved performance to a level acceptable for anything but the most heavily loaded web servers.

1 Introduction

The Internet has spawned its own application domain: multithreaded server applications, capable of interacting with hundreds or thousands of clients simultaneously, are becoming increasingly important. Examples include FTP (File Transfer Protocol), E-Mail transport, DNS (name servers), Usenet News, chat servers, distributed file-sharing, and the most popular of all: HTTP servers. HTTP (HyperText Transfer Protocol [1]) is the protocol used to transport web pages over the Internet.

The HTTP protocol is essentially transaction-based. The client first opens a connection to the server and sends a request message. The server interprets the request and, if the request refers to a valid document on the server, replies to the client sending it the contents of the document. In early versions of the HTTP protocol, the connection between client and server would be closed at this point, requiring the client to open a new connection for each document request. The latest revision of the HTTP protocol allows the connection to remain open for further transfers (known as a keep-alive connection).

The nature of the HTTP protocol imposes certain demands on a server implementation:

- Concurrency is essential, since the server cannot allow itself to be tied up by a slow client, or a client requesting a large document from the server. The choice of concurrency model has the largest impact on the server's performance, as we will discuss in Section 2. The server should be able to handle a large number of clients simultaneously transferring documents without performance being affected.

- For small requests, the server must be able to process the request quickly to avoid tying up resources for too long. This means the latency for a new connection should be as low as possible. This aspect of a server's performance can be measured by firing small requests at it at an increasing rate; at some point, the server's performance will start to drop off as the rate increases beyond the latency and a backlog of connections in progress builds up.

  Another advantage of low latency is that the server is more resistant to denial-of-service type attacks. If it can throw out bogus requests as quickly as possible, then the overall server performance will be impacted less when the server is flooded with requests from a malevolent client.

- Fault tolerance is as important as performance: the server should be able to recover gracefully from errors, and never crash (or if it does, arrange that it restarts itself so that there is a minimal interruption in service). It should also be possible to re-configure the server without taking it down.

So why write a web server in Haskell? Firstly, because we can! It's no bad thing for Haskell if we can compete effectively in important application domains such as web serving. Secondly, Haskell has plenty of relevant attributes to bring to the party:

- Concurrent Haskell [5] provides a lightweight concurrency model that helps to provide the low latency and low overhead for multiple simultaneous clients that is essential for good server performance.

- Recent extensions to support exceptions [6] provide useful facilities for coping with runtime errors.

- Asynchronous Exceptions [3] allow concise implementations of important features such as timeouts, as we shall see in Section 4. Asynchronous exceptions are also useful for allowing the server to respond to external stimulus, such as when the administrator wishes to make changes to the server's configuration (see Section 6).

- We used a wide range of libraries in constructing the web server, including a networking library, a parsing combinator library, an HTML generation library and a POSIX system interface library. None of these libraries are specified as part of the Haskell standard as yet, but all are distributed with the GHC compiler.

This paper describes our web server implementation, focussing on the parts which required extensions to Haskell, in particular the concurrency and exception support. The server implementation performs well, as we shall show in Section 7. It is also reliable - we tested it as an alternate server for the `haskell.org` site, where it ran for a week, collected about 2000 hits during that time, and kept within a memory footprint of 3M. This test unconvered one bug in the HTTP protocol implementation, which has since been fixed. We hope to replace the main `haskell.org` web server (currently Apache) with the Haskell implementation at some point in the future.

2 Concurrency Model

The choice of concurrency model is crucial in the design of a web server. In this section we briefly examine the common models in use and list their advantages and disadvantages, and compare them with Concurrent Haskell's approach.

2.1 Separate Processes

This is the model used by Apache, where each new connection is handled by a new process on the machine. A single top-level process monitors incoming connections and spawns a new worker-process to talk to each client.

Advantages:

- Simple to implement.
- Takes advantage of multiple processors automatically.

Disadvantages:

- Very heavy-weight, in terms of the startup and shutdown cost, context-switch overhead, and memory overhead for each process. (Apache maintains a cache of spare processes in order to reduce the start-up time for new connections).
- Interprocess Communication is hard.

Interprocess communication is required for certain aspects of a web server's operation. For example, there is normally a single log file which records transaction events (see Section 5), and hence multiple threads must cooperate for access to the log file.

2.2 Operating-System Threads

Operating system threads may map onto processes, in which case the overhead will be similar as for separate processes (modern operating systems will share page tables between forked processes in any case), or they may be implemented as light-weight kernel threads, in which case the overhead will be lower.

Advantages:

- Interprocess communication is easier.
- Takes advantage of multiple processors automatically.

Disadvantages:

- Fairly heavy-weight, although possibly lighter than separate processes.
- Writing multithreaded code is harder and more error prone than single threaded code.

2.3 Monolithic Process with I/O multiplexing

Another approach is to implement the desired concurrency directly, foregoing any time-sharing facilities provided by the operating system itself. The single requirement for this approach is to be able to multiplex several I/O channels.

The existing methods for multiplexing I/O in a single process include:

- Use POSIX's `select()` (or equivalently `pol()`) functions. These functions tests multiple file descriptors simultaneous, returning information on which of the descriptors are available for reading or writing. The idea is that the application then performs any available reads and writes (using non-blocking I/O), and then returns to call `select()` on the list of open file descriptors again.

  This approach suffers from the problem that `select()` is $O(n)$, where n is the number of file descriptors being tested, because the application must build up a list of length n to pass to `select()` and the OS must traverse this list to build up the results.

- Asynchronous I/O and POSIX real-time signals. These methods are both ways of alleviating the linear behaviour of `select()`. They are both relatively new features and only a few operating systems support them. Any implementation which uses `select()` can be converted to use asynchronous I/O or realtime signals with relatively little effort.

Advantages of these methods:

- Very fast, especially the latter methods.

Disadvantages:

- A web server is an inherently multi-threaded application, so programming without the concurrency abstraction is bound to be painful.

There exist several web servers which use these methods, and they are currently the fastest servers around.

2.4 User-space threads

User-space threads are essentially an implementation of a thread abstraction inside a single process (or possibly on top of a small number of operating system threads; see later). The programmer gets to write his/her application using the concurrency primitives provided by the language, and the user-space threads implementation will provide the low-level time-sharing and I/O multiplexing support. Several implementations of user-space POSIX threads exist for Unix.

Concurrent Haskell (or at least the implementation in GHC) is also an instance of this model; the Haskell runtime system runs in a single operating system process and multiplexes many Haskell threads. To support multiple Haskell threads performing I/O simultaneously, the runtime system may choose between the I/O multiplexing options described in the previous section. However this is implemented, the choice is invisible to the programmer.

In a way, user-space threads provide the best of both worlds. The concurrency is lightweight, and the programmer doesn't need to be concerned with the details of I/O multiplexing. There is one disadvantage, though: a user-space threads package won't normally be able to take advantage of multiple processors on the host machine. The GHC development team are currently working on an implementation of Concurrent Haskell that doesn't suffer from this deficiency, by using a small number of operating-system threads to share the load.

3 Structure of the web server

3.1 The main loop

The main loop is strikingly simple:

```
acceptConnections
    :: Config -> Socket -> IO ()
acceptConnections conf sock = do
    (handle, remote) <- accept sock
    forkIO (
        catch
           (talk conf handle remote
             `finally` hClose handle)
           (\e -> logError e)
     )
    acceptConnections conf sock
```

`acceptConnections` takes a server configuration and a listening socket, and waits for new connection requests on the socket. When a connection request is received, a new worker thread is forked with `forkIO`, and the main loop goes back to waiting for connections.

The worker thread calls `talk` (the definition of `talk` is given in the next section), which is the main function for communicating in HTTP with a client. The interesting part here is what happens if an exception is raised during `talk`. The `finally` combinator allows strict sequencing to be specified, independent of exceptions:

```
finally :: IO a -> IO b -> IO a
```

This combinator behaves much like `finally` in Java. It performs its first argument, then performs its second argument (even if the first argument raised an exception), then returns the value of the first argument (or re-raises the exception).

In the main loop above, we're using `finally` to ensure that the socket to the client is properly closed down if we encounter an error of any kind, including a bug in our code. Although the runtime system will automatically close files which are determined to be unused, it is beneficial to close them down as early as possible in order to free up the resources associated with the file.

The call to `talk` is also enclosed in a `catch` combinator[1]:

[1] this `catch` differs from the standard `IO.catch` in Haskell; here we are referring to `Exception.catch` from ghc's `Exception` library

```
catch :: IO a -> (Exception -> IO a) -> IO a
```

This combinator performs its first argument, and if an exception is raised, passes it to the second argument (the *exception handler*), otherwise it returns the result. In contrast to `finally`, `catch` specifies an action to be performed only when an exception is raised, whereas `finally` specifies an action which is always to be executed.

The above code uses `catch` to catch any errors and log them to the error log file, which we describe in Section 5.

3.2 HTTP protocol implementation

Serving a request is a simple pipeline:

1. read the request from the socket,
2. parse the request,
3. generate the response,
4. send the response back to the client,
5. if the connection is to be kept alive, return to step 1.

Reading the request from the socket is performed by `getRequest`:

```
getRequest :: Handle -> IO [String]
```

which takes the file handle representing the socket on which communication with the client is taking place, and returns a list of strings, each one being a single line of the request. A typical request looks something like this:

```
GET /index.html HTTP/1.1
Host: www.haskell.org
Date: Wed May 31 11:08:40 GMT 2000
```

The first line gives the command (GET in this case), the name of the object requested, and the version of the HTTP protocol being used by the client. Subsequent lines, termed *headers*, give additional information, and are mostly optional. The server is required to ignore any headers it doesn't understand.

The next stage is to parse the request into a `Request` structure:

```
data Request = Request {
     reqCmd     :: RequestCmd,
     reqURI     :: ReqURI,
     reqHTTPVer :: HTTPVersion,
     reqHeaders :: [RequestHeader]
  }

parseRequest :: Config -> [String]
             -> Either Response Request
```

Note that parsing the request may return a response: this indicates failure, and the response will in most cases be a "Bad Request" response, but may be something more specific.

Next, we generate the response:

```
data Response = Response {
     respCode     :: Int,
     respHeaders  :: [String],
     respCoding   :: [TransferCoding],
     respBody     :: ResponseBody,
     respSendBody :: Bool
   }

data ResponseBody
  = NoBody
  | FileBody Integer{-size-} FilePath
  | HereItIs String

genResponse  :: Config -> Request
             -> IO Response
```

`genResponse` performs a number of checks on the validity of the request, and generates an appropriate response. A valid GET request will result in a response with a `FileBody`, whereas an invalid request will result in an error response of some description, with some automatically generated HTML describing the error in the body (the `HereItIs` body type is for this purpose).

In the common case where the response body consists of an entire file verbatim, the `respBody` component of the `Response` structure doesn't contain the entire file body as a string, rather it contains just the path to the file. This is so that we can use more efficient methods for sending the file to the client than simply converting to the contents to and from a `String`.

The final step is to send the response to the client:

```
sendResponse :: Config -> Handle
             -> Response -> IO ()
```

Pulling all this together, the top-level `talk` function looks something like this:

```
talk :: Config -> Handle -> HostAddress
        -> IO ()
talk conf handle haddr = do
  strs <- getRequest handle
  case parseRequest strs of
    Left resp ->
      sendResponse conf handle resp
    Right req -> do
      resp <- genResponse conf req
      sendResponse conf handle resp
      logAccess req resp haddr
      if (isKeepAlive req)
        then talk conf handle haddr
        else return ()
```

In reality, there is some extra code to deal with catching and logging of errors (Section 5.1) and timeouts (Section 4) in there too. The call to `logAccess` causes an entry to be written to the log file describing the transaction, see Section 5.

4 Timeouts

A web server needs some form of timeout mechanism, so that clients which hang or take an inordinately long time to respond can be disconnected, and the resources associated with the connection freed.

Basically what we need is a generic time-out combinator, with the following type:

```
timeout :: Int    -- timeout in seconds
        -> IO a   -- action to run
        -> IO a   -- action to run on timeout
        -> IO a
```

The application `timeout t a b` should behave as follows: `a` is run until it either completes, or `t` seconds passes. If it completes in time `t`, `timeout` returns the result immediately, otherwise `a` is terminated with an exception and `b` is executed. If `a` (or `b`, in the case of a timeout) raises an exception, then the exception will be propagated by `timeout`.

The `timeout` function has no other side effects, so `timeout`s can be nested arbitrarily.

Thanks to asynchronous exceptions [3], we can implement a `timeout` combinator with the above properties. Note that because the action `a` can be terminated with an asynchronous exception at any time, it should be *exception safe*, that is it must be sure not to leave any mutable data structures in an inconsistent state or leak any resources. In fact, all our code should be written to be exception-safe, because exceptions like stack overflow and heap overflow are delivered asynchronously.

There are two primitives which are useful in writing exception-safe code:

```
block   :: IO a -> IO a
unblock :: IO a -> IO a
```

where `block a` executes `a` with asynchronous exceptions *blocked*, that is any thread wishing to raise an asynchronous exception in the current thread must wait until exceptions are unblocked again. Similarly, `unblock a` unblocks asynchronous exceptions during the execution of `a`. Applications of `block` and `unblock` can be arbitrarily nested. Here's an example of acquiring a lock, where the lock is represented by an `MVar`, `m`, such that the lock will always be released safely if we receive an exception:

```
block (do
  a <- takeMVar m
  (unblock (...))
     `catch`
        (\e -> do putMVar m a; throw e)
  putMVar m a
)
```

Use of combinators such as `finally` (described in the previous section), and `bracket`:

```
bracket :: IO a -> (a -> IO b)
             -> (a -> IO c) -> IO c
```

are also helpful in writing exception-safe code. For example, a simpler way of writing the above locking sequence is

```
bracket (takeMVar m) (putMVar m) (...)
```

The full story on asynchronous exceptions in Haskell, including the implementation of the `timeout` combinator above, can be found in [3].

5 Logging

A web server normally produces log files listing all the requests made and certain information about the response sent by the server. Each entry in the log normally records

- the time the request was received,
- the requestor's address,
- the URL requested,
- the response code (either success or some kind of failure),
- the number of bytes transfered,
- the time taken for the request to complete,
- the client software's type & version,
- the referring URL.

The format of log entries is configurable, and may include other fields from the request or response. There exist standard log entry formats produced by the popular servers, and software available which processes the log files to produce reports. For this reason, we decided that the Haskell web server should be able to produce compatible logs.

A worker thread causes a log entry to be recorded by calling the function `logAccess`:

```
logAccess :: Request
          -> Response
          -> HostAddress
          -> TimeDiff
          -> IO ()
```

passing the request, the response, the address of the client and the time difference between the request being received and completion of the response.

The actual writing of log entries to the file is done by a separate thread. Worker threads communicate with the logging thread via a global unbounded channel, by calling `logAccess`. The logging thread removes items from the channel and manufactures log entries which are then written to a log file.

Placing the logging in a separate thread is a good idea for several reasons:

- It helps reduce the total load on the system, because the worker threads can finish, and hence be garbage collected, before the log entry has been written.

- It means that a thread serving multiple requests can proceed immediately with the next request without waiting for the log entry of the first request to be written.

- The logging thread can batch multiple requests and write them out in one go, which is likely to be more efficient than writing them one at a time.

The logging thread is designed to be fault tolerant: if it receives an exception of any kind, it attempts to restart itself by re-opening the log file for writing, and continuing with the next log request. This behaviour is (ab)used by the main loop, which needs to restart the logging thread whenever it receives a request to re-read the log file.

5.1 Error logging

Logging of server errors is handled in a similar way to logging of requests. A separate thread writes log entries to an error log file, taking log requests from a global channel. Exception handlers are scattered around the main request/response handling code, which catch exceptions and log them with an informative message indicating where the error occurred, before passing the exception on to be handled at the top level.

The error logging thread also restarts itself if it receives an exception (and logs this event to the log file).

5.2 Global variables

In the above description of the logging threads, we mentioned that communication between a worker thread and a logging thread was via a "global channel". How does one define a global channel in Haskell? This is one instance of global mutable variables, which we make use of in several places in the web server. A global `MVar`, for example, can be declared as

```
global_mvar :: MVar String
global_mvar = unsafePerformIO newEmptyMVar
```

While global variables are very convenient in certain cases, there are a few points to bear in mind:

- This use of `unsafePerformIO` really is unsafe, because the program now behaves differently if occurrences of `global_mvar` are replaced with their values, namely `(unsafePerformIO newEmptyMVar)`. In order to stop this happening, we have to circumvent any optimisations in the compiler which replace names with their values. In GHC, this amounts to adding the pragma

  ```
  {-# NoInline global_mvar #-}
  ```

  somewhere in the source code.

- Care must be taken to give a type signature for the global variable and not to declare global mutable variables with polymorphically-typed contents. Type safety is in danger if this rule is broken, because the contents of a polymorphically-typed variable could be extracted and used at any type.

- In a concurrent program, it is important to use `MVar`s instead of just `IORef`s when multiple threads may have access to the variable (an `IORef` is a plain mutable variable, whereas an `MVar` adds synchronisation).

If we observe these rules, however, global mutable variables are a useful concept. We use global mutable variables in the web server in the following places:

- To store the command line options. The program is only started once, so this is a write-once mutable variable.

- To store the channels by which the worker threads can communicate with the logging threads.

- To store the `ThreadId`s of the logging threads, so that the main thread can send them an exception to restart them.

A cleaner, but less efficient, alternative to using global variables would be to use implicit parameters [2]. We haven't investigated this route as yet.

6 Run-time configuration

Our web server is configured by editing a text file, in a similar way to other popular web servers. The syntax of the configuration file is similar to that of Apache's. When the server starts up, it parses the configuration file, and if there are no errors found, immediately starts serving requests.

In the interests of high availability, a web server should preferably also be run-time configurable. For example, when new content is placed on the server, the administrator somehow needs to inform the server that the new content is available and where to find it. It is occasionally necessary to change certain options, or tweak security settings, on a running server.

To take the server down and restart it with the new configuration would be unsatisfactory, because the site would be off-line during the restart. So a running server should be able to re-read its configuration file without interrupting operations. But what about transactions that are already in progress? Should they see the new configuration immediately?

In our server, we take the approach that the new configuration should only take effect for new connections, and existing connections should be allowed to continue using the old settings. This approach avoids a number of problems with changing configuration settings while a request is in progress: for example, if the security settings are changed such that a file being transmitted is no longer available to the client that requested it, should the transfer be terminated? In fact this behaviour is desirable, but a sledgehammer solution is to restart the server altogether if any security settings need to change.

Implementing run-time configuration updates in the Haskell web server turned out to be straightforward: as we've already seen, the key functions in the inner pipeline all take an argument of type `Config`, which contains the current configuration. The configuration is passed into the worker thread when it is created, so when the configuration changes all we need to do is ensure that any new threads receive the new configuration.

The approach we took is to send the main thread an exception when it should re-read the configuration file. This gives us the option of having several ways to force a configuration change:

- A signal on Unix-like operating systems. This is the traditional way to kick a process into re-reading its configuration file, and consists of sending the process a signal from the command line. This method is implemented in our web server as follows: the incoming signal causes a new thread to start, which immediately sends an exception to the main thread. The main thread catches the signal and re-reads the configuration file.
- Implementing a proprietary HTTP command, which the administrator can use to reconfigure the server on-line and remotely. Secure authorisation would certainly be needed if this method were to be used.
- Any other type of inter-process communication provided by the host operating system.

7 Performance Results

In this section we present our preliminary performance results for the Haskell web server.

7.1 Performance tweaks

We made several tweaks to the initial implementation of the server to remove some of the larger performance bottlenecks.

- We replaced the naive file transfer code which used `getContents` and `hPutStr`, with a version which does I/O directly to and from an array of bytes. GHC's `IOExts` library provides simple primitives for doing this.
- By default, GHC's scheduler context switches about 5000 times a second. We reduced this to something more reasonable, 50 times/sec, which made a substantial difference to the results. The reason is that GHC's scheduler currently does a `select` on every context switch to determine which I/O bound threads can be woken up. As discussed in Section 2, `select` is $O(n)$, so reducing the number of times we do it is a win when the system becomes heavily loaded.
- Tweaking the garbage collection settings had some effect, in particular increasing the allocation area size. GHC by default increases

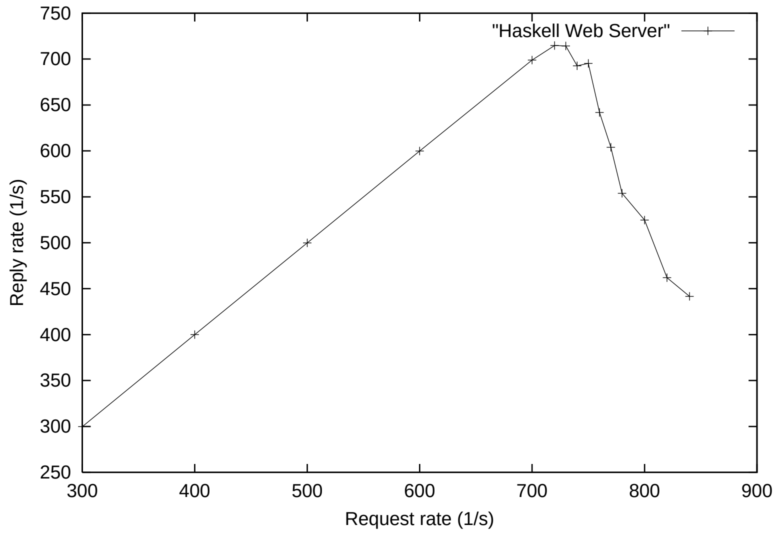

Figure 1: Connection latency results

its heap usage in line with the program's demands, but giving the program more memory from the outset is usually a win, and was in this case.

- Reading a request turned out to be expensive, due to an inefficient implementation of `hGetLine` in GHC's I/O library. Rewriting this function improved performance by 10% or so.

- GHC's I/O library uses a system of finalizers to ensure that the buffers and other resources associated with a file descriptor are freed when the program releases the file handle. Finalizers are normally run in a thread by themselves, but this turned out to be expensive for the web server, since most connections give rise to two finalizers: one for the handle to the socket itself, and one for the file being transfered. We changed the finalisation mechanism to batch finalizers in a single thread after each garbage collection, which led to a small overall performance improvement.

Note that only one of these tweaks, namely the optimisation of the file transfer code, was made to the web server code itself, the rest were tweaks to GHC's runtime system and libraries. Indeed, the web server has been a useful source of insight into performance bottlenecks in GHC's concurrency and I/O support.

7.2 Connection latency

These measurements were made using `httperf` [4], a tool which can be used for generating requests at a specific rate. It is used primarily for determining the rate of connection requests a web server can sustain before performance starts to drop off.

In this test, the server machine was a single-processor PII/450 running Linux 2.2. The client was a separate machine on a local 100Mbit ethernet connection. The total number of requests sent was 4000 in each test. All the requests were for the same 1k file. The timeout on the client was set at 1 second.

A graph of reply rate against requests issued per second is given in Figure 1. This shows clearly how the server keeps up with the client until the request rate rises above the rate that the server can handle without accumulating a backlog (about 710 requests/second), at which point performance begins to decrease sharply. Why does performance decrease so dramatically? Two possible factors are:

- As connections in progress accumulate on the server, the $O(n)$ behaviour of `select()` as used by GHC's scheduler comes into play.
- As the number of threads in the system increases, thus the cost of garbage collection also increases. Garbage collection is necessarily $O(n)$ in the number of live threads, since it must traverse the active thread queues to determine which threads are live.

The server doesn't currently limit the number of of connections in progress, and in fact at a rate of 850 connections/sec the number of concurrent connections observed on the server was over 700 during the test. Setting a limit on the number of concurrent connections would help to flatten the graph after the drop-off point.

On the same hardware, Apache (the most commonly used web server software) tops out at 950 requests/second, and the drop off is less sharp. One reason for the shallower drop off is that Apache limits the number of active connections to 256 by default, with any incoming connections over the limit being simply refused by the operating system.

To put these figures into perspective, the most heavily loaded web servers on the net (eg. `http://www.yahoo.com/`) take an average of about 5000 hits/second, with peaks of probably 10000 hits/second. These sites use collections of identically configured servers with a load-balancing arrangement to spread the requests between the available machines.

However, for most sites on the net the performance turned in by our Haskell Web Server is more than adequate, and there's still plenty of opportunities for improvement: we haven't really made any attempt to optimise the code of the server itself, beyond fixing the slow file transfer.

8 Conclusions

The primary result presented here is that we constructed a web server in Haskell which conforms to the HTTP/1.1 standard (and more) in less than 1500 lines of Haskell (not including library code), and the resulting server performs admirably in real-world conditions. Furthermore, it is fault-tolerant and runs in a constant, and small, amount of memory over a long period of time.

In order to achieve this, we had to make use of a number of extensions to Haskell, the main ones being concurrency and exceptions. We also made use of a large amount of library code, all of which is part of GHC's library collection. The libraries we used include a networking library, a parsing combinator library, an HTML generation library and a POSIX interface library.

References

[1] R. Fielding *et al.* Hypertext transfer protocol - HTTP/1.1, June 1999. RFC 2616.

[2] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, Boston, Massachusetts, January 19–21 2000.

[3] S. Marlow, S. P. Jones, and A. Moran. Asynchronous exceptions in haskell. In *4th International Workshop on High-Level Concurrent Languages*, Montreal, Canada, September 2000.

[4] D. Mosberger and T. Jin. `httperf`—a tool for measuring web server performance. `http://www.hpl.hp.com/personal/David_Mosberger/httperf.html`. Hewlett-Packard Research Labs.

[5] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. POPL'96*, pages 295–308, St. Petersburg, Florida, Jan. 1996. ACM Press.

[6] S. L. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Proc. PLDI'99*, volume 34(5) of *ACM SIGPLAN Notices*, pages 25–36. ACM Press, May 1999.