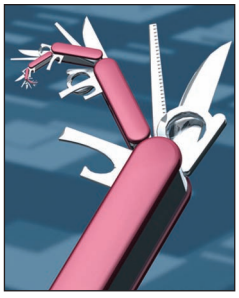
Warp: A Haskell Web Server



Michael Snoyman • *Suite Solutions*

Roughly two years ago, I began work on the Yesod Web framework. I originally intended FastCGI for my deployment strategy, but I also created a simple HTTP server for local testing, creatively named SimpleServer. Because Yesod targets the Web Application Interface (WAI), a standard interface between Haskell Web servers and Web applications, it was easy to switch back ends for testing and production.

It didn't take long before I started getting feature requests for SimpleServer: slowly but surely, features such as chunked transfer encoding and sendfile-based file serving made their way in. The tipping point was when Matt Brown of Soft Mechanics made some minor tweaks to SimpleServer and found that it was already the fastest Web server in Haskell (see Figure 1). After that, he and I made some modest improvements and released the code as Warp.

Very little code in Warp itself is geared toward speed. For the most part, it simply builds on the shoulders of giants — by relying on underlying libraries that perform extremely well, Warp can achieve a lot in fewer than 500 lines of code. Let's explore how Warp uses each of these libraries, what makes them so powerful, and how they fit together.

Glasgow Haskell Compiler

The first library isn't really a library at all: the Glasgow Haskell Compiler (GHC) is the standard Haskell compiler. It has all the optimizations you'd expect of an industrial-strength compiler, such as loop unrolling, extensive inlining, unboxing, and fusion. It even lets users specify their own optimizations via rewrite rules. In addition, it provides a very sophisticated multithreaded runtime. One great thing about this runtime is its lightweight threads. As a result of this feature, Warp simply spawns a new thread for each incoming connection, blissfully unaware of the gymnastics the runtime is performing under the surface.

Part of this abstraction involves converting synchronous Haskell I/O calls into asynchronous system calls. Once again, Warp reaps the benefits by calling simple functions like `recv` and `send`, while GHC does the hard work.

Up through GHC 6.12, this runtime system was based on the `select` system call. This worked well enough for many tasks but didn't scale for Web servers. One big feature of the GHC 7 release was a new event manager, written by Google's Johan Tibell and Serpentine's Bryan O'Sullivan. This new runtime uses different system calls (`epoll`, `kqueue`, and so on) depending on what's available on the target operating system. Additionally, Tibell and O'Sullivan made extensive enhancements to the data structures the manager uses: it now uses a radix trie for storing callbacks and a priority search queue for timeouts.

The end result: Haskell programs can easily scale to thousands of simultaneous connections. Programmers can write their code against a very simple API, spawning new lightweight threads using `forkIO` and calling blocking functions inside them.

Enumerator

A recent move in the Haskell community has been adopting the *enumerator pattern*. This pattern allows for processing streams of data in a deterministic manner. This is especially important for Web servers, which must quickly release scarce resources such as file descriptors.

1089-7801/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

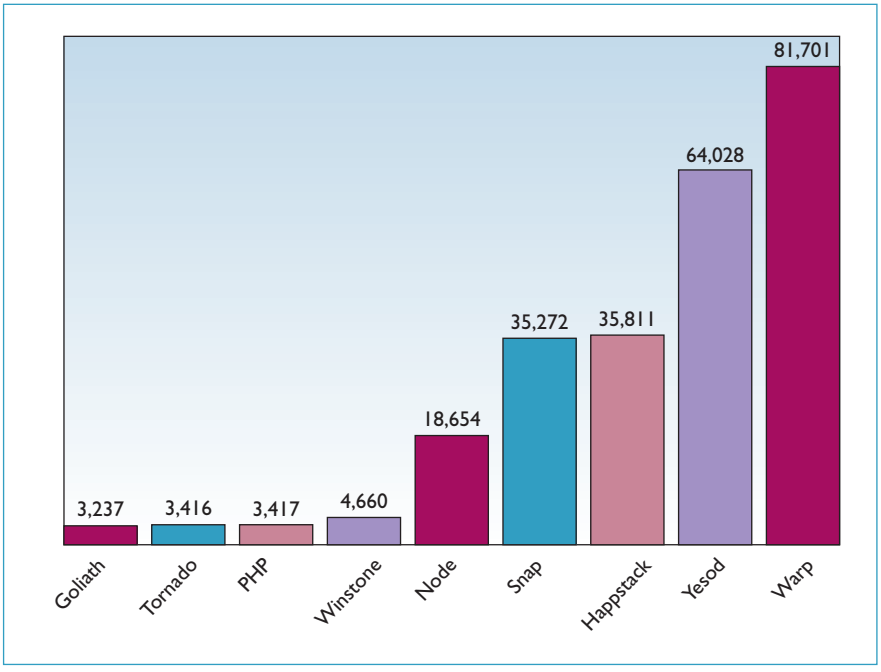


Figure 1. Pong benchmark. Requests/second (higher is better).

John Millikin (unaffiliated) wrote the `enumerator` package that WAI and Warp use. In this package, the central datatype is `Iteratee`. An `Iteratee` is a data consumer, receiving chunks of data and performing some action with them. `Iteratee` is an instance of `Monad`, making it easy to compose two `Iteratees` together to build up more complicated actions. (For those not familiar, a `Monad` is a container that encapsulates a computation's side effects. Haskell programmers can easily combine different monadic values to build up more powerful computations.)

The flip side of `Iteratee` is `Enumerator`, a data producer. An `Enumerator` will feed data into an `Iteratee` until either the `Enumerator` has run out of data or the `Iteratee` no longer accepts more. A simple example of the interaction between these two is file input and output: `enumFile` is an `Enumerator` that reads data from a file and streams it into an `Iteratee`, whereas `iterHandle` is an `Iteratee` that consumes a stream of bytes and sends them to a handle.

A third datatype, an `Enumeratee`, is a combination of `Enumerator` and `Iteratee`: it receives a stream of data from an `Enumerator` and sends a new stream of data to an `Iteratee`.

Warp's entire I/O system is built on top of the `Enumerator` datatype. Once Warp establishes a connection and starts a new handler thread, it produces an `Enumerator` from the client socket and pipes that data into an `Iteratee`. This `Iteratee` is where all request parsing occurs.

`Enumerator`'s built-in chunking behavior also works perfectly for Warp as well. The `Enumerator` optimizes the size of its requested buffers, currently set at 4,096 bytes. The consuming `Iteratee`, on the other hand, has no concept of these chunks' size. Instead, it simply consumes as many bytes as it wants. If there isn't enough buffered content to complete an operation (for example, the chunk terminated in the middle of an HTTP header), then control automatically returns to the `Enumerator` to provide more output. If too much data was provided, the remainder is left in the `Enumerator` to be consumed by the next action. It will either be part of the request body and sent to the application, or will be part of the next request.

`Enumeratees` also play an important role in Warp. They ensure that the application consumes the entire request body before continuing with the next request, and that the application doesn't consume more bytes than it should for the request body. They also convert the response body from a stream of `Builders` (discussed next) to a stream of bytes with chunked transfer encoding applied.

Blaze-Builder

The simplest way to represent a string in Haskell is as a list of Unicode characters. This has two major performance issues: it's expensive to append data to a list, and the representation of the list has a lot of overhead. Historically, two different solutions have existed, one solving each issue:

- Use difference lists instead of actual lists during data construction, and produce only the final output list at the end. This exploits the fact that appending to a difference list is an O(1) operation.
- Represent our data using a packed format such as `ByteString` or the newer `Text` datatype.

The `blaze-html` package, by Jasper Van der Jeugt of Ghent University and Simon Meier of ETH Zurich, sought to solve both issues during HTML content construction. The idea is to work around the central concept of a `Builder`, a value that knows how it should fill up a memory buffer. Internally, a `Builder` is a difference list of these buffer-filling actions. Combining these two points, we end up with a packed representation of data with efficient append operations. And, just as important, we're guaranteed that the bytes will be copied precisely once into our final buffer.

It quickly became apparent that the `Builder` abstraction would be useful outside the context of HTML generation. The Yesod Web framework immediately used it for generating CSS, JavaScript, and JSON. Meier split off the `Builder` datatype and its associated functions into a separate `blaze-builder` package.

WAI and Warp rely heavily on `blaze-builder` for constructing responses. Applications always send their response bodies to the server in the form of `Builders`. This lets Warp efficiently append the body to the response headers, meaning that, for many common responses, Warp uses only a single memory buffer and makes a single system call. As a nice finishing touch, `blaze-builder` provides a helper function to automatically prepend the length of each chunk of an HTTP response when using chunked transfer encoding. This function has taken care of the complicated logic of concatenating `Builders` to an optimal size and backtracking to fill in the chunk size in the header, and it's available for all Haskell HTTP servers to use.

Blaze-Builder-Enumerator

At one point, I needed to write a program at work to modify XML files. Because it was simply modifying attributes, this was a perfect case for a streaming algorithm, and thus a great use case for `Enumerators`. Millikin had already written a parsing wrapper for the C language `libxml` library, but no method existed for generating output.

The simplest approach would be to convert each XML event into a `ByteString`. However, this would involve creating a lot of small buffers. A better approach would be to use `Builders`, but consuming the entire stream of `Builders`, concatenating them, and then writing to a file would involve keeping the entire body in memory, something I wanted to avoid.

Instead, I ended up writing an `Enumeratee` that would take a stream of `Builders` and use them to fill up buffers. When a buffer filled, the `Enumeratee` would wrap it in a `ByteString` and send it down the pipeline to the `Iteratee`. This meant that the code produced optimally sized `ByteStrings`, with minimal buffer copying, and used constant memory. Meier has since taken the code, improved it, and released it as `blaze-builder-enumerator`.

It turns out that the exact same requirements exist when writing a Web server. The application can give the server chunks of data of any size, and the server wants to concatenate these into optimally sized buffers to minimize system-call overhead, without using large amounts of memory or performing multiple buffer copies.

In Warp, when the application returns a `ResponseEnumerator` response, the flow control will pass back and forth between the server and the `Enumerator`. The `Enumerator` will feed chunks of `Builders` to the server. The server then fills up a memory buffer using those `Builders`. Once the buffer is filled, the server will send its contents over the socket and release the buffer. This means the data is copied precisely once from the `Builder` into the final buffer. Additionally, the server must allocate only a single buffer, so memory usage is constant.

Web Application Interface

The WAI is a low-level interface between Web applications and back ends in Haskell. It's generic enough to support standalone servers such as Warp, as well as options like the Common Gateway Interface (CGI), FastCGI, and development servers that automatically reinterpret your code during development. Warp is the premiere WAI back end.

The WAI concept is very simple: an application is a function that takes a request and returns a response. The `Request` datatype contains information such as the requested path, query strings, request headers, and remote host/port. One thing noticeably lacking from this list is the request body. To understand why, consider the following type signature:

```
type Application = Request →
  Iteratee ByteString IO Response
```

The `Application` returns its `Response` inside an `Iteratee`, so it consumes the request body from there. As mentioned previously, Warp performs all its operations inside the `Iteratee` monad; this means that calling the `Application` is simply another step in that process. The beauty of the `Enumerator` approach is that these actions compose together so easily.

Three types of responses exist, represented by different data constructors. `ResponseFile` contains a status code, a list of response headers, and the path to a file. This allows back ends like Warp to use an efficient `sendfile` system call for sending the file contents to the client.

`ResponseBuilder` contains a status code, a list of response headers, and a single `Builder` value. This is the most commonly used response type. In most programming languages, this would require storing the entire response in memory. Haskell, on the other hand, uses lazy evaluation by default, meaning the value will compute on demand. So, we can efficiently encode very large responses as this single value, and the application will consume memory only as needed.

The most interesting type of response is `ResponseEnumerator`, which lets an application produce responses while interleaving impure actions. One example usage would be to stream a large database response to the client. Although we could do this using `ResponseBuilder`, it

would require reading the entire database response into memory and then sending it. With `ResponseEnumerator`, control will pass between an application and the back end. As soon as Warp has enough data to fill a buffer, it will immediately send the data to the client and then release the memory the previous pieces have consumed.

Request Parsing

The Warp team was able to implement a clear, concise, safe, and efficient request parser, thanks in large part to Haskell's high-quality `ByteString` library (from Don Stewart of Galois, Duncan Coutts of Oxford University, and David Roundy of Oregon State University). The library provides a high-level interface to C byte arrays with an elegant mix of expressiveness and efficiency. We can use `ByteStrings` in much the same way as linked lists, through versions of many idiomatic list functions familiar to functional programmers. They also interface directly with standard C I/O facilities with zero conversion. The API functions are bounds-checked by default, although unchecked versions are also available. Warp uses these in several cases when the operation is statically known to be safe.

An HTTP request begins with a request line and an unspecified number of header lines. The end of the headers is indicated by a blank line. Lines are delimited by pairs of carriage return and linefeed characters, and headers are key-value pairs separated by colons. Scanning the input for new lines and colons is performed efficiently via `memchr`. The request parser then extracts data using copy-free substring functions.

Although the protocol syntax is very simple, a few minor complications exist. A single read block can contain multiple lines, and lines can span multiple read blocks. We also have several exceptional conditions to look out for: the client might take too long to send us data or close the connection without sending the terminal blank line. We also need to prevent attackers from filling up memory by sending an infinitely long header.

GHC's exceptions and lightweight threads let us abstract away the timeout and unexpected end-of-file cases into a single blocking function call. Our parser is responsible only for ensuring that the header size is under the allowed maximum.

The parser is implemented as a recursive `Iteratee` of four arguments: the current accumulated header size, two difference lists (one accumulating segments of the current header line, and one for completed header lines), and the current input buffer. In addition to providing O(1) appends, using difference lists here preserves tail call optimization.

Each recursive call consumes some data, appends it to the current line, and increments the header size. Once the parser reaches a line terminal, if the completed line isn't empty, it's appended to the list of headers. It then creates a new difference list for the next line, and the function recurses. If the completed line is empty, we've reached the end of the header. Any data remaining in the input buffer goes back to the `Enumeratee`, and the function returns.

Timeout Handling

A relatively recent attack vector for Web servers is a *slowloris attack*: an attacker opens as many connections as possible to a server and sends trickles of data across them in an attempt to exhaust the server's connection pool. This attack can work especially well because it requires so few resources from the attacker. The standard response is to introduce timeouts: if a client doesn't send any data after a specified amount of time, disconnect the socket.

The first version of Warp used the timeout handling code included with GHC. Unfortunately, this was not a very good fit; it didn't scale well, and, even worse, introduced deadlocks into the code. (GHC has since fixed this bug, but most users are still running affected versions.) So, Warp needed a more elegant solution.

This was another opportunity for the Haskell Web development community to shine: Gregory Collins of Google and Jeremy Shaw of See-Reason Partners (who work on the Snap and Happstack frameworks, respectively) had already been collaborating on more efficient timeout code. They had a great start, but the initial code was slower than we hoped for. I made two changes to their approach:

- The original code used `MVars`. This is a thread-safe, mutable variable that would usually be the perfect fit for our use case. Unfortunately, the locking overhead was simply too much. I switched to using an `IORef` instead. Unlike an `MVar`, an `IORef` is simply a mutable variable without any locking. However, it provides an atomic modify operation, which takes advantage of Haskell's referential transparency to avoid race conditions without locking.
- A lot of complexity was involved in managing a mutable, thread-safe hash table for storing the timeout information. Because we know that all functional programmers only really know about lists, I decided to try them out here, with much success.

The entire timeout library is less than 70 lines of code. It works by creating a timeout manager thread and an `IORef` holding a list of handles. Each handle contains an action to perform on timeout (killing the appropriate thread) and its