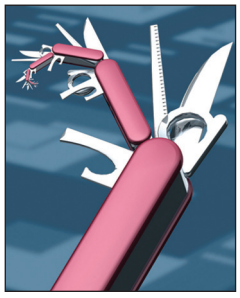

Yaws: Yet Another Web Server

Steve Vinoski • *Verivue*

The modestly named Yaws — "Yet Another Web Server" — is an open source Erlang Web server known for its reliability, stability, and scalability. It started roughly a decade ago as the brainchild of legendary Erlang programmer Claes "Klacke" Wikström, who also invented several important Erlang features, including Erlang term storage (ets), Distributed Erlang, the Mnesia database, and the Erlang bit syntax. Yaws is a general-purpose HTTP 1.1 Web server consisting of a relatively simple core surrounded by implementations of various features, such as response streaming, AJAX support, websockets support, Common Gateway Interface (CGI) and Fast CGI (FCGI) support, and application embedding. While it's perfectly capable of performing regular file serving, Yaws is most useful for applications that generate and serve dynamic content.

Configure, Make, Install

To get started with Yaws, you can either obtain an official release from its primary website, http://yaws.hyber.org, or you can retrieve the latest sources from the Yaws github.com source repository:

```
git clone git://github.com/klacke/yaws.git
```

You can build Yaws in two different ways: either using GNU autotools and make, or via the rebar Erlang build tool (see https://github.com/basho/rebar). To use the autotools approach:

```
autoconf
./configure
make && sudo make install
```

This will build Yaws and by default install it into `/usr/local/bin/yaws`, with Erlang beam bytecode files and other parts of the Yaws system installed elsewhere under `/usr/local`. Command-line options for the generated configure script let you choose alternative installation locations. This approach is suitable for running Yaws as a stand-alone Web server. Alternatively, you can build Yaws using rebar:

```
rebar compile
```

The result of this approach, which builds Yaws and installs it within its own build area, can also run stand-alone but is best-suited for applications that embed Yaws as an internal Web server.

Runtime Configuration

Yaws configuration controls details such as where it looks for files being served, what ports it listens on, and what applications it loads to handle requests. Two configuration portions exist. *Server configurations* apply to individual virtual servers. A single Yaws instance can host multiple virtual servers on the same or different IP addresses. Server configuration controls features such as on which port and IP address the server listens for requests, the root path in the filesystem from which Yaws serves files, the types of scripts the server can execute, and the application modules, or *appmods*, that are assigned to various URI paths. Appmods are very useful for dynamic Yaws Web applications, so I'll present more details about them later.

Global configuration applies to the running Yaws instance as a whole. It specifies directories where logs are deposited, from which Erlang bytecode is loaded, and from which include files are retrieved for use in dynamically compiled application code. It also sets size and timeout

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

limits for the Yaws internal file cache.

The default configuration, which Yaws reads from a file at startup, sets up a regular HTTP server listening on port 8000 and a Secure Sockets Layer (SSL) server listening on port 4443. No changes to the default configuration are needed to have a working system, but of course you're free to edit the configuration to your liking. Extensive documentation is available for all global and server configuration settings.

Internally, Yaws represents global configuration using an Erlang record named `gconf` and server configurations using Erlang `sconf` records. These records are defined in a public include file for use by Yaws applications, especially those that start Yaws as an embedded Web server. Embedded applications can't rely on Yaws reading its configuration from a file and must instead feed a valid configuration via `gconf` and `sconf` record instances to Yaws when starting it as an embedded server.

File Serving

Yaws primarily targets applications that generate dynamic content, but we couldn't consider it a general-purpose Web server if it couldn't serve files. Each server configuration tells Yaws where to find the document root for that server, and Yaws then uses request URI paths to find requested files relative to that root. There's nothing special about Yaws's file serving capabilities except that it's alone among Erlang Web servers in using the `sendfile` operating system system call where available to make file delivery as efficient as possible. However, it likely won't hold this distinction forever, given that at the time of this writing, the `sendfile` capability was being extracted from Yaws and refactored for inclusion in a future release of Erlang and its accompanying Open Telecom Platform (OTP) utility framework. Once it appears there, it will be generally available to all Erlang Web servers and other applications.

```
<html>
<erl>
out(Arg) ->
  {ok, Title} = yaws_api:queryvar(Arg, "title"),
  {ok, Name} = yaws_api:queryvar(Arg, "name"),
  {ehtml,
    [{head, [],
      [{title, [], Title}]},
     {body, [],
      [{p, [], ["Hello, ", Name]}]}]}.
</erl>
</html>
```

Figure 1. The example.yaws file. This file shows a Yaws `out/1` Erlang function embedded between `<erl>` and `</erl>` tags. When a client requests the example.yaws page, Yaws invokes the `out/1` function, compiling it first if necessary, and substitutes its return value in place of the `<erl>` and `</erl>` tags and everything between them. Yaws then returns the resulting HTML page to the client.

Dynamic Applications

Because Yaws primarily targets dynamic applications, it provides several useful features for them. One such feature is "ehtml," which is regular HTML represented as Erlang terms. For example, here's some simple HTML:

```
<head><title>Hello World
  </title></head>
<body><p>Hello, World!</p>
  </body>
```

And here is the same content represented in ehtml:

```
{ehtml,
  [{head, [],
    [{title, [], "Hello World"}]},
   {body, [],
    [{p, [], "Hello, World!"}]}]}
```

As the example shows, ehtml constructs are quite similar to their HTML counterparts but, assuming familiarity with Erlang syntax, are easier to read. Each HTML tag is represented as an Erlang atom, and each construct is generally a 3-tuple comprising a tag, a list of attributes, and a body list. The head tag shown here, for example, has an empty attribute list and a body list consisting of the title 3-tuple, which itself also has an empty attribute list and a body list which is the title string (in Erlang, strings are just lists of characters). Applications can create ehtml constructs more easily than strings of HTML, and ehtml is less error-prone because the Erlang compiler enforces syntactical correctness of the tuples and lists.

A related dynamic application feature is the ability to embed Erlang code within HTML. When a request arrives for a file with a ".yaws" suffix, Yaws processes any Erlang code it finds within the file between `<erl>` and `</erl>` tags, replacing the tags and the code between them with the result of executing the code. Such code, which is expected to reside within a function named `out/1` (where the "/1" indicates the function *arity*, or number of arguments), is compiled and also cached for future invocations.

For example, consider the somewhat contrived example.yaws file in Figure 1. If a client requests this resource from the Yaws server, the actual document served depends on

```
out(Arg) ->
  Uri = yaws_api:request_url(Arg),
  NewId = create_new_id(),
  Segs = [Uri#url.path, NewId],
  NewPath = string:join(Segs, "/"),
  NewUri = Uri#url{path = NewPath},
  NewLoc = yaws_api:format_url(NewUri),
  Json = json2:encode(
    {struct, [{"location", NewLoc}]}),
  [{status, 201},
   {header, {"Location", NewLoc}},
   {content, "application/json", Json}].
```

Figure 2. Application module. This appmod `out/1` function returns a response suitable for a `POST` request creating a new resource. It first obtains the target URI and augments a copy of it with an identifier for the new resource. It then creates a JSON object with a "location" key to store the new resource URI, and also returns the new URI as the value of the response `Location` header. The result specifies HTTP status and the `Location` header, and includes the JSON object string as the response body.

values of the query parameters "title" and "name" appearing within the URI used to access the resource. Within the code, the two calls to the `yaws_api:queryvar/2` function return the values of the two query parameters, which are then used to complete the ehtml returned from the `out/1` function. Accessing the resource via the URI `http://example.org/example.yaws?title=Example&name=Joe` substitutes the text "Example" for the title and the name "Joe" for the variable part of the paragraph text. This example, though contrived (and potentially unsafe, given that the query parameter values are used unescaped), shows how an application can call any Erlang functions to dynamically create any content it wants to return to its clients.

Arg Data

You probably noticed the parameter called `Arg` passed to the `out/1` function. This is an instance of the Yaws `arg` record, a structure that encapsulates everything Yaws knows about the current request, including

- the client connection socket,
- the client IP address and port,
- request headers,
- various sections of the path portion of the request URI,
- the request body (such as for HTTP POST, PUT, and PATCH), if any, and
- query parameters, if any.

The `arg` instance is key to programming Yaws applications. Once Yaws passes it to your code, the code can then pass it back to various Yaws functions to extract information from it to help fulfill the request.

Appmods

Not all Web resources can be represented only in HTML, of course, so not all applications can use ehtml or ".yaws" pages. Yaws applications wanting to deal with common formats such as XML and JSON can do so via appmods. An appmod is conceptually quite simple: it's an Erlang module that exports an `out/1` function. You configure Yaws to associate the module with a portion of a URI path. When the configured server receives a request containing that path, Yaws invokes the appmod's `out/1` function, passing it an `arg` instance representing the request. For example, registering an appmod in a given server under the path "/" means that it will receive all requests sent to that server.

The result of the `out/1` function typically supplies the status, headers, and body for the client response. For example, the appmod in Figure 2 returns a response suitable for a `POST` request creating a new resource.

This `out/1` function creates a new resource and a URI to identify it. It first extracts the target URI from the request `Arg`, which returns a Yaws `url` record instance. It then calls the `create_new_id/0` function to generate a new identifier to use in the new URI (the details of `create_new_id/0` are unimportant here). It then copies the target URI record but sets its path component to that of the new URI path string. To create the response body, it uses the Yaws `json2` module to create a JSON object whose "location" key contains the URI string for the new resource. Finally, it returns a list of tuples specifying the HTTP response: a 201 status code indicating the creation of a new resource, a `Location` HTTP header specifying the URI for the new resource, and the content body as a JSON string specified with the "application/json" media type. Before writing the response to the client connection socket, Yaws augments this with other typical headers, such as `Date`, `Server`, and `Content-Length`.

Appmods are infinitely flexible. To handle requests and form responses, they can call into databases, invoke backend services, implement caches, or communicate with replicas on other Erlang nodes. You can implement them as regular functions or as calls into instances of standard Erlang behaviors such as `gen_server` and `gen_fsm`.

Yapps

A single drawback to appmods, though, is that to be deployed, they require configuration changes to associate the appmod with a URI path. If an application is developed

by a single developer or small team, this isn't a big problem. However, consider the fact that Erlang/OTP is equipped with a modular application packaging and deployment system that allows applications from independent developers to be deployed together within a single Erlang node. Under this system, it would be nice if Web applications could be stitched together from independently developed component applications and could be deployed into already-running Yaws instances without requiring configuration file changes.

A *yapp*, short for "Yaws application," is similar to an appmod but encompasses a whole Erlang application rather than just a single module. This means the way the application starts, its supervision tree, and the way it's upgraded at runtime are all independent of Yaws. To enable yapp registration, Yaws provides a yapp module that allows registration under URI paths via a webpage without the need for configuration file changes. But consistent with appmods, each yapp simply provides one or more `out/1` functions that Yaws calls when it decides to dispatch a request to the yapp.

But Wait! There's More!

Features like file serving, ehtml, ".yaws" pages, appmods, and yapps aren't all that Yaws provides. It also offers various other features that this column space doesn't allow me to describe in detail, including but not limited to

- *Arg rewriting*. Applications can register modules that can examine and modify the request arg record instances before Yaws dispatches requests for them, which allows applications to affect how that dispatching occurs.
- *CGI and FCGI support*. This lets Yaws fulfill client requests via calls to external programs and scripting languages, such as PHP and Perl, that support these protocols.
- *Websockets*. Yaws clients can send an HTTP upgrade request to ask Yaws to switch the protocol over to websockets, which allows HTTP-independent bidirectional communication between client and server. At the time of this writing, the websockets specification was still in draft form and not finalized, so Yaws support is currently considered experimental and subject to change. Once the specification is completed, we'll update Yaws to conform to it.
- *Response streaming*. This allows Yaws applications to essentially take over the client connection from Yaws to stream responses to the client, which is useful, for example, for long-polling applications (Comet applications) or for serving video streams. The application `out/1` function returns an indication to Yaws that it wants to stream a response to the client from a given Erlang process (which is like a very lightweight thread), after which Yaws sends a message to that process to tell it that it now owns the client socket. Once the application's process is finished with the socket, it sends a message back to Yaws to return socket ownership.
- *JSON-RPC and SOAP*. These two protocols are tunneled through HTTP, letting clients interact with remote resources in a fashion different than what HTTP allows. JSON-RPC and SOAP both allow remote procedure calls (RPCs) to remote resources, and SOAP also allows interaction with remote resources via non-RPC XML message exchange. Both approaches are fundamentally flawed in that they treat HTTP as a transport protocol rather than the application protocol that it actually is, thereby bypassing and eliminating many of the decoupling, evolvability, and scalability benefits it provides. Despite these flaws, some Yaws users still see these protocols as useful, so Yaws supports them.
- *Reverse proxy*. This experimental code allows Yaws to be configured to proxy requests from clients and send them to entirely separate Web origin servers that Yaws effectively hides from clients.
- *Embedding*. As mentioned earlier, applications can embed Yaws as an internal Web server rather than just being appmods or other components that a stand-alone Yaws controls.

A number of these features were donated by users who required the functionality for their production systems, and they've been present in the Yaws source code repository for several years.

Performance

At this point, most articles like this one would provide graphs and charts showing the results of executing benchmarks against whatever Web server the article describes. Despite their popularity, such benchmarks are never as generally useful as they might seem. They're typically contrived, rarely represent real-world loads, are often nearly impossible to reproduce, depend highly on the tuning of the underlying platform, and are sometimes even rigged to make the subject Web server appear superior. While there's no doubt that benchmarks are a very useful tool for developers of servers and frameworks to measure, assess, and improve the throughput and latency of their own code, too many users misinterpret such benchmarks by mistakenly equating the fastest with being the best.

Writing a simple Web server in Erlang isn't difficult because the support for HTTP header parsing is

built into the Erlang runtime and is activated by setting a flag on a socket representing a client connection. The Erlang runtime also provides socket event handling. While this is quite convenient for applications, it also means that well-written Erlang Web servers don't greatly differ from each other as far as performance goes because they're all using these same facilities. At first glance, this might seem problematic, but it really isn't, for at least the following reasons:

- Developers tend to choose Erlang for its reliability, stability, concurrency support, and ease of development. Naturally, they want reasonable performance, but they're not necessarily seeking to give up these other important characteristics to maximize performance.
- Developers normally don't choose an Erlang Web server for static file serving, but rather, as mentioned earlier, for dynamic applications. For such applications, the server-side application itself, and not the underlying Web server, is often the bottleneck in terms of throughput and latency because of calls it makes into databases or backend services to obtain response data.
- Let's face it — while everyone likes to think their applications require the utmost in performance and scalability, few actually do. Most dynamic applications will never even approach the performance limits of Yaws or any other Erlang Web server.

That being said, my observations of production systems show Yaws performance and scalability to be quite good, especially considering the number of features it provides. It holds up well against other Erlang Web frameworks that advertise themselves as being lightweight, including some that lack robust support for HTTP 1.1 or fail to provide important operational features such as access logging. But don't take my word for it; if you're interested in using Yaws and want to be sure it performs well, I advise you to benchmark it yourself, as only you know best what your application will look like and what sorts of systems it will need to deploy on, integrate with, and depend on for data. I also advise you to follow the sage advice of Mark Nottingham of Yahoo as far as HTTP load testing goes; on his blog he provides an excellent set of rules for how to best obtain useful and meaningful results from such testing (see www.mnot.net/blog/2011/05/18/http_benchmark_rules).

Yaws Going Forward

Klacke and I actively maintain and enhance Yaws, occasionally accepting patches for fixes and new features from users. We watch developments in areas such as websockets and work to keep Yaws up-to-date and also keep an eye out for problem areas in the code or documentation that need attention. Klacke typically creates two or three official releases per year, and users are always welcome to obtain the Yaws source code from github.com and build it themselves. Users communicate with Klacke, other users, and me using the Yaws mailing list (see https://lists.sourceforge.net/lists/listinfo/erlyaws-list).

Our ultimate goal is to keep Yaws as reliable, stable, well-performing, and feature-rich as its long-time users have come to expect.

Steve Vinoski is a member of the technical staff at Verivue in Westford, Mass. He's a senior member of IEEE and a member of the ACM. You can read Vinoski's blog at http://steve.vinoski.net/blog/ and contact him at vinoski@ieee.org.