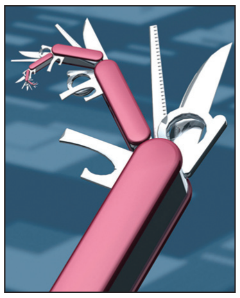
Editor: Steve Vinoski • vinoski@ieee.org



Server-Sent Events with Yaws

Steve Vinoski • Basho Technologies

Traditionally, a Web client sends an HTTP request to a Web server and waits for a response. For many Web clients, the HTTP request-response protocol is perfect for loading either static or dynamically created pages and for posting request data, such as shopping orders, to the server and waiting for either a confirmation or error response. Think about how many Web applications you use on a daily basis that fit comfortably within the HTTP request-response model.

Of course, not all Web applications work well with HTTP's request-response approach. Some work best with notifications or updates from the server, where clients receive messages without having to first send requests to initiate server replies. Event-oriented communication models are nothing new; they've been used in all sorts of distributed computing systems for decades. In such systems, a client typically registers its network endpoint information with a server along with some indication of the types or classes of events and notifications it's interested in receiving. When the server detects a relevant event, it consults its client registry and sends a notification to each client registered for events of that type.

Getting Notified

Early Web clients wanting notifications had little choice but to fake it via polling. At first glance, such an approach might seem horribly inefficient, with myriad clients potentially hammering a beleaguered server with requests for polled resources only to discover no changes to those resources. Fortunately, HTTP provides ways to avoid poor designs of this nature. For example, servers can return last-modified times and entity tags (etags) for polled resources, enabling a client to perform conditional `GET` requests. With such a request, if nothing has changed since the client's last retrieval, the server returns a status of `304 Not Modified`, thereby avoiding expending network resources to return the same resource data again and again. Servers can also use HTTP response headers to indicate that their polled resources can be cached by clients or intermediaries, thereby reducing their own request loads by allowing clients to reuse responses or get their data directly from intermediate caches.

Even though HTTP polling can be surprisingly efficient, it still has drawbacks, the most obvious being how to determine the best polling frequency. If events occur at a rate much greater than a client's polling frequency, many events might no longer be relevant by the time the client gets around to requesting them. On the other hand, if clients poll too frequently, even servers employing savvy caching and conditional `GET` support can be kept overly busy. Finding a happy medium requires either tuning clients manually and then hoping server event frequency doesn't vary much, or writing smart clients that track event resource modification times and dynamically adjust their polling frequencies to match. Both approaches seem brittle and difficult.

To avoid polling, over the years applications have tried other workarounds. For example, servers supporting a technique called "long polling" avoid replying to polling clients immediately if no events await them, holding replies until new events arrive. Other approaches — such as HTTP server push (also known as HTTP streaming) and pushlets — work similarly, with servers holding client connections open after receiving requests in order to push new events back to them as parts of what end up being very long-duration responses.

IC-16-05-Funw.indd 2 7/14/12 4:24 PM

Fortunately, modern Web clients have more options than just polling and nonstandard pseudo-polling techniques. For example, at the other end of the spectrum is the WebSocket Protocol (RFC 6455; http://tools.ietf.org/html/rfc6455), which lets clients and servers agree to convert their HTTP connection to use an entirely different and usually custom application protocol. WebSocket runs over the same TCP connection the client and server originally established for HTTP communication, but rather than being restricted to the request-response model or to HTTP messages, WebSocket is bidirectional, allowing either end to initiate a message; clients and servers have the flexibility to exchange any data and messages in any format they agree on.

WebSocket's flexibility is a blessing and a curse, for at least a couple of reasons. First, while it can clearly supplant polling and easily support event-oriented applications, it requires the client and server to agree on yet another application protocol to run over WebSocket between them. Most WebSocket-based applications today tend to exchange simple JavaScript Object Notation (JSON) messages using basic custom application protocols, but how advanced they'll become in the future is hard to predict. The fact that the Web has scaled as it has owes much to the architectural constraints built into HTTP (RFC 2616; www.ietf.org/rfc/rfc2616.txt); ad hoc application protocols running over WebSocket are unlikely to scale anywhere nearly as well. In the long run, however, one good outcome might be that WebSocket opens the door to several standardized application protocols designed with specific constraints that purposefully differ from HTTP's constraints, letting Web applications intentionally make different tradeoffs depending on what they're trying to do. Such standardization would likely take years to complete.

Second, WebSocket imposes additional interoperation, infrastructure, and implementation details on both client and server above and beyond HTTP. It requires clients and servers to know how to make and service connection upgrade requests, respectively, and it also requires both ends to know how to format and parse WebSocket messages. Fortunately, the WebSocket specification isn't overly complicated, and numerous servers, clients, and Web frameworks already support it (including the Yaws Web server discussed later); still, it imposes nontrivial interoperation and implementation taxes that will likely mean some servers, frameworks, and clients will never make the jump to supporting it.

For Web applications that just want to employ a simple and efficient server-based notification model, a happy medium exists between polling and WebSocket: Server-Sent Events (SSE). This approach — which as of this writing is a W3C working draft (www.w3.org/TR/eventsource/) — lets servers send event data to clients using regular HTTP based on techniques pioneered by HTTP server push and pushlets. Despite its status as a working draft, numerous servers and clients already use the technique, likely due to how easily Web server developers can implement it, as we'll see later.

A client wishing to receive SSE-style events from a server sends it a regular HTTP request, specifying a resource just as with any other request, but also specifying `text/event-stream` as the desired content type of the response. Assuming the server supports such event streams, it replies with regular HTTP response headers specifying the content type as `text/event-stream`. It then holds the connection open without finishing the response, and as it obtains event data for the client, it sends them as simple text messages. Just as with the older HTTP server push and pushlets approaches, each such message is part of the same open-ended HTTP response. The messages can optionally have specific event names and can also have identifiers that let a disconnected client reconnect and tell the server the ID of the last event it saw, thus letting the server restart the event stream where the client left off.

There are two elements to the simplicity of SSE: it's based on regular HTTP, and it's text-based, making it easy to create and parse events.

Yaws and Server-Sent Events

Yaws is an open source Erlang Web server hosted on Github (see http://github.com/klacke/yaws and http://yaws.hyber.org). Its creator, Claes "Klacke" Wikström, started the project in late 2001, and it's been under active development ever since. It's a fully stocked system providing a long list of features both new and old, including HTTP 1.1, forward and reverse proxies, response streaming, WebSocket, SOAP, haXe, JSON-RPC 2.0, SSL, and request rewriting. It's widely used within the Erlang community as well as by others who aren't Erlang developers, and is best known for its great performance and high reliability.

In June 2012, I added support for SSE to Yaws in version 1.94. The implementation, which is very straightforward as we'll see later, sits on top of the Yaws response streaming capability. This capability exploits Erlang's lightweight processes[1] to let an application-supplied process take control of a socket from Yaws and use it to communicate directly with the client. The following steps describe how this works:

1. An application registers a callback with Yaws for a given URL path.
2. When Yaws sees a request including that path, it dispatches the request to the application's callback.

```
out(A) ->
  case (A#arg.req)#http_request.method of
    'GET' ->
      case yaws_api:get_header(A#arg.headers, accept) of
        undefined ->
          {status, 406};
        Accept ->
          case string:str(Accept, "text/event-stream") of
            0 ->
              {status, 406};
            _ ->
              {ok, Pid} = gen_server:start(?MODULE,
                                           [A], []),
              yaws_sse:headers(Pid)
          end
      end;
    _ ->
      [{status, 405},
       {header, {"Allow", "GET"}}]
  end.
```

Figure 1. The `server_sent_events:out/1` function handles incoming requests for the event stream. It first verifies that the client performed a `GET` request and supplied an `Accept` header specifying the type "text/event-stream." If so, it starts the event streaming process via `gen_server:start/3` and returns the appropriate response headers via `yaws_sse:headers/1`.

3. The callback validates the request and either starts a new Erlang process to handle it or retrieves an existing handler process, perhaps from a pool.
4. The callback returns to Yaws, supplying a list of HTTP response headers. Last in the list is a tuple consisting of the `streamcontent_from_pid` directive and the request handler's process ID.
5. Yaws sends the HTTP response headers and then gives control of the client socket to the stream handler process.
6. Once the handler decides it's completed the response, it returns control to Yaws, optionally closing the socket before doing so.

With this capability, supporting SSE isn't difficult: the handler process merely needs to watch for its application-specific events, format them according to the W3C specification, and send them over the socket to the client.

Yaws Example

Yaws supplies an example that shows how to use SSE to show and update the server date and time on a simple webpage. Assuming you have Yaws version 1.94 or later, you can find it in the Yaws distribution in the files `examples/src/server_sent_events.erl` and `www/server_sent_events.html`. The former handles the initial client request and supplies the handler process for event streaming, whereas the latter supplies the client-side HTML along with the JavaScript code that handles the event stream.

The `server_sent_events` module is a Yaws application module, or "appmod" (see http://yaws.hyber.org/appmods.yaws). It exports a function named `out/1` taking a single argument (hence the "/1", which indicates the function arity), which is a Yaws `#arg` record containing all the details of each incoming request. In the Yaws configuration file, we register the appmod on the URL path "/sse", which is the HTTP event resource the client will request to receive the event stream. Any client request made to that path causes Yaws to invoke the `server_sent_events:out/1` function, passing an `#arg` instance containing all the details of the request, including the incoming HTTP headers. The `out/1` function, shown in Figure 1, is fairly simple; it just ensures that the client submitted a `GET` with a requested response content type of `text/event-stream` and, if so, creates a new event handler process and returns its ID along with the appropriate HTTP headers to Yaws.

The event handler process is also implemented in the `server_sent_events` module, using the Erlang/OTP (Open Telecom Platform) `gen_server` framework, or "behavior" in Erlang parlance. (OTP is a set of frameworks and libraries that are part of the Erlang open source distribution. It's useful in a variety of domains, not just telecom.) As its name implies, `gen_server` is useful for implementing server processes; it enables these processes to reply to incoming requests and to maintain state. For this example, we use it to hold onto the client socket and to maintain a timer that indicates when to fire new events.

The event handler supplies several functions that `gen_server` requires, but only two are important for the example. Figure 2a shows a function named `handle_info/2` that handles the message from Yaws indicating that it has handed control of the client socket to our `gen_server` event streamer process; Figure 2b shows a different `handle_info/2` function that handles the recurring `tick` message from the timer.

Both the `out/1` function and the timer handler `handle_info/2` function use functions in the `yaws_sse` module, which provides the core support for SSE-compliant events. It supplies several functions for formatting event IDs, names, and data, as the W3C working draft requires,

IC-16-05-Funw.indd 4 7/14/12 4:24 PM

and also furnishes the `send_events/2` function to send formatted event data to the client. This example uses only event data, and doesn't need either event names or IDs.

Using the command-line tool `curl`, you can issue a `GET` request to the example event resource to see the details of the server response. The following command assumes you have Yaws version 1.94 or newer installed and running on your local machine, listening on port 8000:

```
curl -D /dev/tty \
  -H 'Accept: text/event-stream' \
  http://localhost:8000/sse
```

On the first line of the command, the `-D /dev/tty` option (which assumes you're running the command on Mac OS X, Linux, or some other UNIX system) directs the response headers to the terminal. On the second line, the `-H` option sets the `Accept` HTTP header for the request to the required `text/event-stream` MIME type. The final line specifies the URL of the event stream resource.

The server response appears as follows (but note that some lines hereare artificially wrapped due to space constraints):

```
HTTP/1.1 200 OK
Connection: close
Server: Yaws 1.94
Cache-Control: no-cache
Date: Fri, 29 Jun 2012
   13:47:07 GMT
Content-Type: text/event-stream

data:Fri, 29 Jun 2012
   13:47:08 GMT

data:Fri, 29 Jun 2012
   13:47:09 GMT

data:Fri, 29 Jun 2012
   13:47:10 GMT

data:Fri, 29 Jun 2012
   13:47:11 GMT
```

```
handle_info({ok, YawsPid}, State) ->
    {ok, Timer} = timer:send_interval(1000, self(), tick),
    {noreply, State#state{yaws_pid=YawsPid, timer=Timer}};
```
(a)

```
handle_info(tick, #state{sock=Socket}=State) ->
    Time = erlang:localtime(),
    Data = yaws_sse:data(httpd_util:rfc1123_date(Time)),
    ok = yaws_sse:send_events(Socket, Data),
    {noreply, State};
```
(b)

Figure 2. Example `handle_info 2` functions. (a) When Yaws sends the event streamer process the `{ok, YawsPid}` message to indicate that it can start sending events on the client socket, the `server_sent_events:handle_info/2` function starts a timer to send a message to the event streamer every second. It then stores the reference to the timer in its `gen_server` state. (b) This instance of `handle_info/2` handles the tick message from the recurring timer. When the message arrives, this function gets the local time, converts it to a GMT date string, and calls the `yaws_sse:data/1` function to send the string as an event to the client.

First, the `200 OK` response shows that the `GET` was successful. As the W3C working draft recommends, the `Cache-Control` header is set to `no-cache` to prevent the client or any intermediaries from caching the response. The last header, `Content-Type`, is set to `text/event-stream` as the client requested.

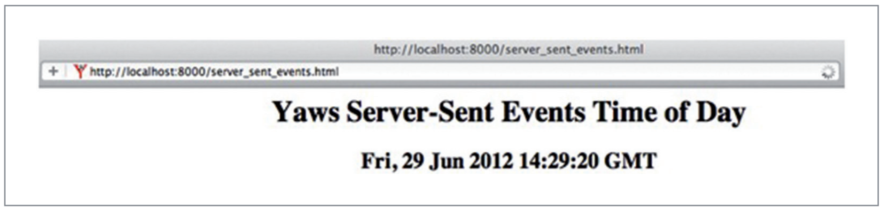
The remainder of the response shows four events. Each line of each event is preceded by the "data:" marker to let receivers distinguish event data from event IDs and names. In this example, each event consists of only a single line and a newline, but the W3C working draft also allows multiline events. Each whole event, whether just a single line or multiple lines, is also terminated by a newline, hence the blank line between each event in the response.

Note that the example response shown here might be misleading for two reasons: First, it makes the response appear as if it all arrives at once, but it doesn't. The headers arrive first, and then the first time-of-day event arrives after a one-second pause. The difference between the value of the `Date` header and the first event shows this pause. After that, each subsequent event arrives individually, one second after the previous event.

Second, due to space considerations, the output shows only four events, but if you run the command yourself, you'll see that the actual event stream continues for as long as we allow it. We eventually have to kill or interrupt the `curl` command to close the connection and stop the stream.

As the `curl` example shows, SSE can be used in any application using HTTP for communication, but it's also instructive to view the resource using a Web browser that supports event streams. Using a recent version of Safari, Chrome, Firefox, or Opera, try visiting the URL http://localhost:8000/server_sent_events.html if you have Yaws running locally, or http://yaws.hyber.org/server_sent_events.html to load the example directly from the Yaws website. If your browser supports event streams, you'll see a simple page that looks like the screenshot in Figure 3, except the time of day will dynamically update on the page every second.

The `server_sent_events.html` page contains some simple HTML to provide the static text, and also loads


http://localhost:8000/server_sent_events.html
http://localhost:8000/server_sent_events.html

Yaws Server-Sent Events Time of Day

Fri, 29 Jun 2012 14:29:20 GMT

Figure 3. Visiting the `server_sent_events` *example in a browser that supports event streams results in a webpage in which the time of day is dynamically updated once per second.*

JavaScript to handle the event stream and dynamically display it. The JavaScript code is straightforward:

```
var es = new EventSource('/sse');
es.onmessage = function(event) {
  var h2 = document.
     querySelector('#date-time');
  h2.innerHTML = event.data;
}
```

First, the code creates an instance of an `EventSource` class, which knows how to parse SSE-style events and make their data available to your client application. The single argument to the `EventSource` constructor, "/sse," is the URL path for the event source resource, just as we saw with the `curl` example. The code then sets a callback function on the event source, which is called whenever the event source receives an event message. The callback function receives one argument: an `event` object. The function simply selects the HTML element in the page with the ID "date-time," which is a header element, and changes its text to that of the event data, which is the time-of-day event string accessible via the `data` property of the `event` object. Every time the server sends a new time-of-day event string, the callback function runs, dynamically updating the time of day displayed on the webpage.

W3C SSE make sending notifications to your Web applications — for example, social network updates, changes in values of financial instruments, or cluster monitoring alerts — uncomplicated and painless. They might not be as flexible or powerful as WebSocket but that doesn't mean they're not effective. They work over plain old HTTP, they're easy for applications to create and receive, and numerous Web browsers and servers already support them. The event names and event ID features allow even mobile applications running on devices with spotty service and limited battery life to efficiently retrieve only the events they missed while disconnected. And with solid building blocks such as Erlang's lightweight process model and the response streaming capability of Yaws, implementing server support for SSE is especially straightforward, making it easy for Yaws applications to use it for their Web clients.

Acknowledgments

Thanks to Stefan Tilkov of innoQ for his comments on a draft of this article.

Reference

1. J. Armstrong, *Programming Erlang: Software for a Concurrent World*, The Pragmatic Bookshelf, 2007.

Steve Vinoski is an architect at Basho Technologies in Cambridge, Massachusetts. 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 or on Twitter at @stevevinoski.

cn *Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.*