Community Ruby Elixir Phoenix

Websocket Shootout: Clojure, C++, Elixir, Go, NodeJS, and Ruby

by Jack Christensen on September 1, 2016

When a web application has a requirement for real time bidirectional communication, websockets are the natural choice. But what tools should be used to build a websocket server? Performance is important, but so is the development process. A performance benchmark alone is not enough. We also need to consider the ease of development. This shootout compares a simple websocket server implemented idiomatically in Clojure, C++, Elixir, Go, NodeJS, and Ruby.

The Test

The servers all implement an extremely simple protocol with only two messages: `echo` and `broadcast`. An echo is returned to the sending client. A broadcast is sent to all connected clients. When the broadcast is completed a broadcast result message is returned to the sender. Messages are encoded in JSON. Both messages take a payload value that should be delivered to the appropriate destination.

Example broadcast message:

```
{"type":"broadcast","payload":{"foo": "bar"}}
```

For the platforms with low level websocket implementations the above message would work directly. For platforms with higher level abstractions such as Phoenix and Rails the message must be encoded to be compatible with their message standards.

Meet the Contestants

Let's start by examining the interesting parts of each contestant.

Clojure

The Clojure server is built on the HTTP Kit web server. The interesting parts are in server.clj.

When a client connects they are stored in an atom of channels. This gives us concurrency safety, which is important since Clojure will handle requests in parallel.

```
(defonce channels (atom #{}))

(defn connect! [channel]
  (log/info "channel open")
  (swap! channels conj channel))

(defn disconnect! [channel status]
  (log/info "channel closed:" status)
  (swap! channels disj channel))
```

Broadcasting is quite simple. The message is sent to all channels, then the result is sent to the sender.

```
(defn broadcast [ch payload]
  (doseq [channel @channels]
    (send! channel (json/encode {:type "broadcast" :payload payload})))
  (send! ch (json/encode {:type "broadcastResult" :payload payload})))
```

Including some uninteresting setup and HTTP routing (but excluding the CLI runner), the core Clojure server weighs in at under 50 LOC.

C++

The C++ server uses websocketpp which in turn relies on boost for its websocket server infrastructure. It is easily the most verbose and complicated code base. The server class is in server.cpp and server.h.

Multithreading is explicit. Multiple threads are started that all execute the websocketpp server `run` method.

```
void server::run(int threadCount) {
  boost::thread_group tg;

  for (int i = 0; i < threadCount; i++) {
    tg.add_thread(new boost::thread(&websocketpp::server<websocketpp::config::asio>::run, &wspp_server));
  }
```

```
  tg.join_all();
}
```

Similar to the Clojure implementation, the live connections are stored in a collection.

```
void server::on_open(websocketpp::connection_hdl hdl) {
  boost::lock_guard<boost::shared_mutex> lock(conns_mutex);
  conns.insert(hdl);
}

void server::on_close(websocketpp::connection_hdl hdl) {
  boost::lock_guard<boost::shared_mutex> lock(conns_mutex);
  conns.erase(hdl);
}
```

To be thread-safe a lock must be acquired before adding or removing a connection. Creating the `boost::lock_guard` object locks `conns_mutex`. The destructor unlocks it automatically when the function returns.

Given this is C++, the syntax for defining the set of connections is *interesting*.

```
class server {
  // ...
  std::set<websocketpp::connection_hdl, std::owner_less<websocketpp::connection_hdl>> conns;
};
```

This is a `std::set` templatized for `websocketpp::connection_hdl`. Since `websocketpp::connection_hdl` is a `std::shared_ptr` it also needs to be templatized on the comparison predicate for the set.

The core broadcast function is relatively straightforward, though verbose.

```
void server::broadcast(websocketpp::connection_hdl src_hdl, const Json::Value &src_msg) {
  Json::Value dst_msg;
  dst_msg["type"] = "broadcast";
  dst_msg["payload"] = src_msg["payload"];
  auto dst_msg_str = json_to_string(dst_msg);

  boost::shared_lock_guard<boost::shared_mutex> lock(conns_mutex);

  for (auto hdl : conns) {
    wspp_server.send(hdl, dst_msg_str, websocketpp::frame::opcode::text);
  }

  Json::Value result_msg;
  result_msg["type"] = "broadcastResult";
  result_msg["payload"] = src_msg["payload"];
  result_msg["listenCount"] = int(conns.size());
  wspp_server.send(src_hdl, json_to_string(result_msg), websocketpp::frame::opcode::text);
}
```

First, it builds the JSON data to send to all clients. Then it acquires a shared lock on `conns` and sends the message to all of them. Finally, it builds and sends the broadcast result message.

The core websocket server class is around 140 LOC.

Elixir

The Elixir server uses the Phoenix framework. The relevant code is in room_channel.ex.

Phoenix has a channel abstraction built-in so there is no need to manually manage the collection of connected clients. It automatically uses JSON as the transport which further simplifies the code.

```
defmodule PhoenixSocket.RoomChannel do
  use Phoenix.Channel

  def join("room:lobby", _message, socket) do
    {:ok, socket}
  end

  def handle_in("echo", message, socket) do
    resp = %{body: message["body"], type: "echo"}
    {:reply, {:ok, resp}, socket}
  end

  def handle_in("broadcast", message, socket) do
    bcast = %{body: message["body"], type: "broadcast"}
    broadcast! socket, "broadcast", bcast
    resp = %{body: message["body"], type: "broadcastResult"}
    {:reply, {:ok, resp}, socket}
  end
end
```

Pattern matching makes handling the client requests elegant. The entire relevant channel code in only about 20 LOC.

Go

The Go server directly uses the `net/http` and `golang.org/x/net/websocket` libraries for websockets. The websocket handler is defined in handler.go.

There is no higher level abstraction other than the connection, so similar to C++ a map is used to store all connected clients. As with C++, a mutex is used for concurrency safety.

```
func (h *benchHandler) Accept(ws *websocket.Conn) {
  // ...

  h.mutex.Lock()
  h.conns[ws] = struct{}{}
  h.mutex.Unlock()

  // ...
}
```

The broadcast method is fairly simple. Take a read lock on the connection mutex. Then send the message to every connection. Finally, unlock the mutex and send the broadcast result to the sender.

```
func (h *benchHandler) broadcast(ws *websocket.Conn, payload interface{}) error {
  result := BroadcastResult{Type: "broadcastResult", Payload: payload}

  h.mutex.RLock()

  for c, _ := range h.conns {
    if err := websocket.JSON.Send(c, &WsMsg{Type: "broadcast", Payload: payload}); err == nil {
      result.ListenerCount += 1
    }
  }

  h.mutex.RUnlock()

  return websocket.JSON.Send(ws, &result)
}
```

Boilerplate such as explicit package importing, explicit error handling, and using strong types instead of untyped maps combine to push the Go websocket code to nearly 100 LOC.

Javascript / NodeJS

The Javascript implementation uses NodeJS and websockets/ws. The entire server is contained in index.js.

The `websockets/ws` server keeps track of connected clients automatically. The code is so simple it needs little explanation.

```
var WebSocketServer = require('ws').Server;
var wss             = new WebSocketServer({ port: 3334 });

function echo(ws, payload) {
  ws.send(JSON.stringify({type: "echo", payload: payload}));
}

function broadcast(ws, payload) {
  var msg = JSON.stringify({type: "broadcast", payload: payload});
  wss.clients.forEach(function each(client) {
    client.send(msg);
  });

  ws.send(JSON.stringify({type: "broadcastResult", payload: payload}));
}

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(message) {
    var msg = JSON.parse(message);
    switch (msg.type) {
      case "echo":
        echo(ws, msg.payload);
        break;
      case "broadcast":
        broadcast(ws, msg.payload);
        break;
      default:
        console.log("unknown message type: %s", message);
    }
  });
});
```

The overall application is easily the shortest: only 31 LOC in one file.

Ruby / Rails

Rails 5 introduced ActionCable, a websocket abstraction very similar to the one in Phoenix.

Like Phoenix it automatically manages the collection of connected clients and handles JSON parsing and serialization. The code is similar in structure and size.

```
class BenchmarkChannel < ApplicationCable::Channel
  def subscribed
    Rails.logger.info "a client subscribed: #{id}"
    stream_from id
    stream_from "all"
  end

  def echo(data)
    ActionCable.server.broadcast id, data
  end
```

```
  def broadcast(data)
    ActionCable.server.broadcast "all", data
    data["action"] = "broadcastResult"
    ActionCable.server.broadcast id, data
  end
end
```

At about 20 LOC for the websocket handling, Rails is in the same class as Phoenix and NodeJS.

Benchmarks

The goal of these tests is to see how a server performs under heavy load, not merely testing if it can handle a large number of mostly idle connections.

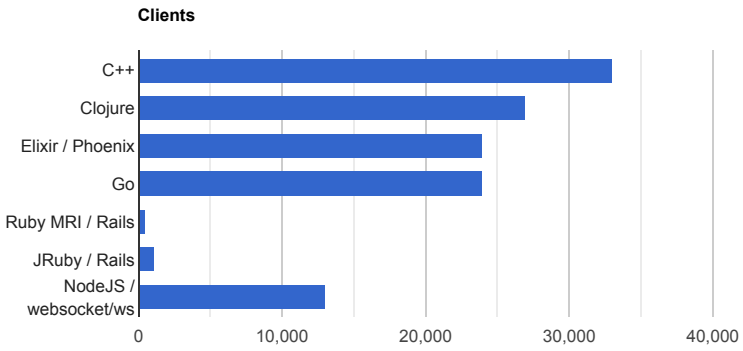
As part of this comparison a benchmark tool `websocket-bench` was built to test the performance of these websocket servers. `websocket-bench` is designed to find how many connections a server can handle while providing an acceptable level of performance. For example, given the requirement that with 10 simultaneous broadcasts in progress at least 95% of broadcasts must be completed within 250ms, how many connections can the server handle?

Here is an example benchmark run:

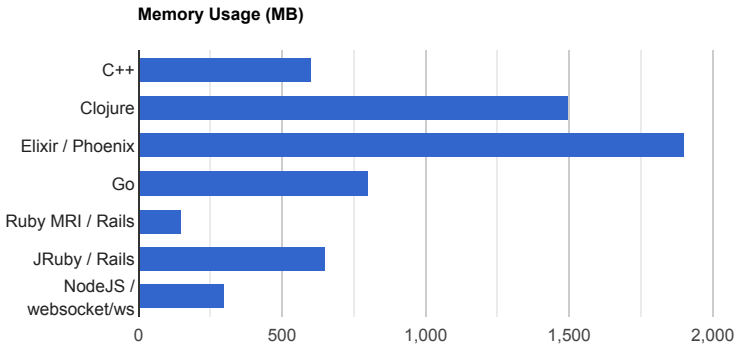
```
$ bin/websocket-bench broadcast ws://earth.local:3334/ws --concurrent 10 --sample-size 100 --step-size 1000 --limit-percentile 95 --limit-rtt 250ms
clients:  1000    95per-rtt:  47ms    min-rtt:   9ms    median-rtt:  20ms    max-rtt:  66ms
clients:  2000    95per-rtt:  87ms    min-rtt:   9ms    median-rtt:  43ms    max-rtt: 105ms
clients:  3000    95per-rtt: 121ms    min-rtt:  21ms    median-rtt:  58ms    max-rtt: 201ms
clients:  4000    95per-rtt: 163ms    min-rtt:  30ms    median-rtt:  76ms    max-rtt: 325ms
clients:  5000    95per-rtt: 184ms    min-rtt:  37ms    median-rtt:  95ms    max-rtt: 298ms
```

The above benchmark starts by connecting 1000 websocket clients to ws://earth.local:3334/ws. Then it sends 100 broadcast requests with a concurrency of 10. It increases by 1000 clients at a time until the 95th percentile round-trip time exceeds 250ms. In this case the server can meet its performance requirements with up to 5000 clients.

These results are from running the server on one machine and the benchmark tool as another. Both machines are bare metal 4ghz i7 4790Ks with 16GB of RAM running Ubuntu 16.04 connected via GB ethernet. The tests were run with a concurrency of 4 and a 95th percentile requirement of 500ms round-trip time. Tests were run multiple times and the best results were recorded.



Memory usage can also be a factor when running in constrained environments.



Unsurprisingly, C++ is at the top of the performance chart by a substantial margin. It also is most efficient in memory usage for the number of connections. However, the C++ server is also the most verbose and the most complex implementation. The language is an enormous multi-paradigm conglomeration that includes everything from the low-level memory management, raw pointers, and inline assembly to classes with multiple inheritance, templates, lambdas, and exceptions. A developer also must delve into

compile flags, makefiles, long compile times and parsing arcane error messages. On the plus side, deployment can be simple as you can compile a project down to a single binary. If you absolutely need the most performance this is where it's at, but be prepared to pay for it in development time and difficulty finding skilled developers.

At 82% the performance of the leader, Clojure proves that very high level languages can be very fast. It is a JVM hosted language, so it can benefit from the rich assortment of Java libraries. Clojure is in the Lisp family, so it may be a challege to developers who have never worked in that style of language. But once that hurdle is overcome, Clojure can be quick to develop and the code is very concise. Deployment depends on the JVM, but beyond that an entire project can be contained in a `jar` file so deployment is relatively easy. If you have developers with Clojure skills, Clojure is a great choice.

Elixir is in third place at 73% of the speed of C++. Elixir attaches a friendly Ruby-style syntax to the functional and highly concurrent Erlang VM. The code is clear, concise, and easily read even by developers who do not know Elixir. Phoenix adds a channel abstraction that makes websockets very easy to work with. Best practices for deployment are still being determined, see Gatling for Hashrocket's take on Elixir deployment. On the downside, Elixir developers are still rare and the language is new and still changing (for example, date and time types were just standardized in Elixir 1.3 which was released in June 2016). Elixir is a solid option for websocket servers.

Go tied Elixir for third place for performance. It uses about half the memory as Clojure and Elixir. Go has a different type of simplicity than the other languages on this list. It eschews any magic or hidden behavior. This has lead to a very simple and clear but verbose language. Go's implementation of CSP makes it far easier to reason about concurrent systems than event-driven systems. Go compiles to a static binary which makes deployment simple. As with the last couple options, Go is a good choice for websocket servers.

Trailing further behind is NodeJS at 39%. NodeJS performance is hampered by its single-threaded architecture, but given that fact it performs very well. The NodeJS server was the smallest overall in lines of code and extremely quick and easy to write. Javascript is the *lingua franca* for web development, so NodeJS is probably the easiest platform for which to hire or train developers.

At less than 2% the performance of C++, Rails running on Ruby MRI simply cannot compete on web socket performance. It's role is supplementing traditional Rails applications that only need websockets for a small number of concurrent clients. In that particular case, avoiding another technology stack just for websockets can be a win. The winner for Ruby performance testing was JRuby. Though still far behind the other contestants, JRuby more than doubled the performance of MRI. JRuby should definitely be considered for any Rails deployment.

Conclusions

When it comes to websockets, Clojure, Elixir, and Go are all choices that balance great performance with ease of development. NodeJS is in a lower performance tier, but is still good enough for many uses. Rails can only be recommended when part of an existing Rails application with few concurrent clients. C++ offers the best performance, but it is hard to recommend due to the complexity and difficulty of development.

Code and complete benchmark results for this shootout are at Github.