The `websockets` Package

Bryan W. Lewis
blewis@illposed.net

October 31, 2011

1 Introduction

HTML 5 websockets define an efficient socket-like communication protocol for the web. The `websockets` package is a native websocket implementation for R that supports most of the draft IETF protocols in use today by web browsers. The `websockets` package is especially well-suited to interaction between R and web scripting languages like Javascript. Multiple simultaneous websocket server and client connections are supported.

The `websockets` package has few dependencies and, written mostly in R, is easily portable. It lets Javascript and other scripts embedded in web pages directly interact with R, bypassing traditional middleware layers like .NET, Java, and web servers normally used for such interaction. And in some cases, websockets can be much more efficient than traditional Ajax schemes for interacting with clients over web protocls.

The `websockets` package provides three primary capabilities:

1. An websocket service.
2. An websocket client.
3. A basic HTTP service.

This guide illustrates each capability with simple examples.

2 Running an R `websockets` server, step by step

The `websockets` package includes a server function that can initiate and respond to websocket and HTTP events over a network connection (websockets are an extension of standard HTTP). All R/Websocket server applications share the following basic recpie:

1. Load the library.
2. Initialize a websocket server with `create_server`.
3. Set callback functions that will respond to desired events.
4. Service the server's socket interface with `service`, often in an event loop.
5. Shutdown the server and delete the server environment when done.

We outline the steps with examples below.

2.1 Load the library

```
library("websockets")
```

The library depends on the `caTools`, `bitops` and a recent version of the `digest` package. It suggests that the `RJSONIO` library be installed, as it is quite useful to have available when interacting with Javascript.

2.2 Initialize a websocket server with `create_server`

The R/Websocket service is initialized by a call to the `create_server` function. (The initialization method called `createContext` from older versions of the package is still supported.) The function takes two arguments, a network port to listen on, and an optional function closure to service standard HTTP requests (described in greater detail below). The `create_server` function returns an environment that stores data associated with the newly created server. "Callback" functions may be assigned that respond to specific websocket and generic http events. Here is an example that creates a websocket server on the default port of 7681:

```
server = create_server()
```

Multuple websocket servers may be defined, but they must use distinct ports.

The websocket server will respond directly to any websocket client request. For convenience, the server may optionally also service basic, non-websocket HTTP reuests. For example, the basic package demo available from `demo('websockets')` serves clients the file `basic.html` located in the package installation path. Additional examples are provided below. See the Rook package for an alternate comprehensive R web service.

2.3 Set callback functions to respond to events

Clients may connect to the websocket service immediately after the server is initialized. The server may write data to or close client connections at any time. However, one must define functions to respond to incoming client events.

Each websocket server instance supports the following incoming events:

`established`: Occurs when a websocket client connection is successfully negotiated.

`closed`: Occurs when a client websocket connection has been closed.

`receive`: Occurs when data is received from a connection.

R functions may be defined to handle some, all, or none of the above event types. Such functions are termed "callbacks."

The `set_callback` function may be used to define a callback function in the server environment returned by `create_server`. (It simply assigns the functions in that environment.)

The `receive` callback function must take precisely three parameters that are filled in by the library with values corresponding to an event that invokes the callback. The required parameters are: `DATA`: A vector of type raw that holds any incoming data associated with the event. (It may be of length zero if the event does not have any data to report.); `WS`: The websocket client associated with the event, represented as an R list; `HEADER`: Header data returned by newer protocol versions, or NULL for protocol version 00. The complete frame header is returned as described in the IETF Draft http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-17.

The `closed` and `receive` functions must each take one argument, a `WS` websocket client associated with the event, represented as an R list.

The following example `established` function sends a text message to each newly-established connection:

```
f = function(WS) {
  websocket_write("Hello there!", WS)
}
set_callback("established", f, server)
```

Here is an example `receive` callback that receives data from a client connection and simply echoes it back to the client:

```
g = function(DATA, WS, ...) {
  websocket_write(DATA, WS)
}
setCallback("receive", g, server)
```

2.4 Accept requests from web clients

Javascript and other web script clients can very easily interact with the R websockets library directly from most browsers. The listing below presents a very basic example client web page that includes Javascript code to open a connection to a local websocket server running on port 7681. See the demo scripts in the the package installation path for more complete examples.

```
<html><body>
<script>
socket = new WebSocket("ws://localhost:7681", "chat");
try {
  socket.onmessage = function got_packet(msg) {
    document.getElementById("output").textContent = msg.data;
  }
catch(ex) {document.getElementById("output").textContent = "Error: " + ex;}
</script>
<div id="output"> SOCKET DATA APPEARS HERE </div>
</body></html>
```

Note: The websockets package presently ignores the sub-protocol ("chat" in the above example).

2.5 Service the socket interface with service

Incoming websocket events are queued. The service function processes events on a first-come, first-served basis. The service function processes each event by invoking the appropriate callback function. It returns after a configurable time out if there are no events to service. Events may be processed indefinitely by evaluating the service function in a loop, for example:

```
while(TRUE)
{
  service(server)
}
```

The service function timeout value prevents the R session from spinning and consuming lots of CPU time. See the service help page for more information.

2.6 Sending data to clients

The websocket_write and websocket_broadcast functions are used to send data to connected clients. The websocket_broadcast function emulates a true broadcast by sending data in a loop to all connected websocket clients associated with the specified server.

The websocket_write function may be used at any time to send data to a specific websocket client. Each websocket server environment returned by the create_server function maintains a list of connected client sockets in the variable client_sockets. Each client socket is in turn represented by an R list. The following example assumes that the server environment has been initialized and contains at least one connected client:

```
websocket_write("Hello", server$client_sockets[[1]])
```

Note the use of the double bracket indexing operator to select a single list element from the client_sockets list.

2.7 Close the server when done

Servers should be closed when done as follows:

```
websocket_close(server)
```

2.8 HTTP convenience functions

The websockets package includes two convenience function closures for servicing basic HTTP requests to non-websocket clients: static_file_service and static_text_service. The functions take either a file name or text string that contains an HTML web page, respectively, and issue a well-formed HTTP 200 response to the requesting client. They are intended to be used in the webpage argument to the create_server function. These functions may be used to furnish web browser clients with an HTML page that contains Javascript code to establish a websocket connection to R.

The following example defines a basic web page:

```
content='<html><body>
  <script>
  socket = new WebSocket("ws://localhost:7681", "chat");
  try {
    socket.onmessage = function got_packet(msg) {
      document.getElementById("output").textContent = msg.data;
    }
  catch(ex) {document.getElementById("output").textContent = "Error: " + ex;}
  </script>
  <div id="output"> SOCKET DATA APPEARS HERE </div>
  </body></html>
'

server = create_server(webpage=static_text_service(content))
```

The web page text will be issued to any client making an HTTP GET request. To serve content from files instead, use the `static_file_service` function. Additionally, the `static_file_service` function checks to see if the file has been updated and always uses a fresh version.

Note that both convenience functions mostly ignore the GET RESOURCE and all other GET or POST request parameters. They always only return the specified HTML content. See the section on using the package as a generic HTTP service below for more comprehensive examples.

3 R as a websocket client

The `websockets` package includes the `websocket` function for creating clients that can interact with other websocket services. It supports protocol versions 00 and newer protocols up to at least version 08 (version 00 and 08 are most widely used by web browsers at the time of this writing).

The `websocket` function returns an environment similar to the `create_server` function, with a single list element in the enclosed `client_sockets` variable corresponding to the client. Set callback functions on the new client context to handle websocket events just as outlined above for websocket servers. And use the `websocket_write` function exactly as outlined about to write data through the client connection to the connected server.

The following example connects to a publicly available websocket echo server (using the 00 protocol).

```
> library(websockets)
> client = websocket("ws://echo.websocket.org", port=80)
> set_callback("receive", function(DATA,WS,HEADER) cat(rawToChar(DATA)), client)

> websocket_write("Testing, testing", client)
[1] 1

> service(client)
Testing, testing

>websocket_close(client)
```

See the `websocket` man page for more information.

4 Using the `websockets` package as a basic web server

The websockets package includes functions that may be used to define a basic (non-websocket) web service. See the Rook package for an alternate comprehensive R web service.

The webpage argument to the create_server function specifies a generic HTTP callback function. The callback function must take two arguments, socket and header, that represent a low-level client socket connection and the full HTTP header. Users are free to define arbitrary HTTP handler functions to respond to the incoming generic HTTP request. The websockets package provides two helper functions for use in HTTP callbacks:

- **http_vars(socket, header)**
  Parse the HTTP header for GET or POST variables, returning them in a list.
- **http_response(socket, status, content_type, content)**
  Write a well-formed HTTP response back to the client socket, closing the connection when done.

The following example illustrates a basic HTTP-only service that asks the user for a stock ticker symbol and produces a historic plot of prices looked up from Yahoo Finance using the quantmod package.

```
library("websockets")
library("caTools")
library("quantmod")

httpd = function(socket, header) {
  body = "<html><body><form>Ticker: <input type='text' name='symbol'/></form>"
  vars = http_vars(socket, header)
  if(!is.null(vars)) {
    getSymbols(vars$symbol)
    f = tempfile()
    jpeg(file=f, quality=100, width=650, heigh=450)
    chartSeries(get(vars$symbol), name=vars$symbol, TA=c(addVo(),addBBands()))
    dev.off()
    img = base64encode(readBin(f,what="raw",n=1e6))
    unlink(f)
    body = paste(body,"<br/><img src='\n", img,"'</img>")
  }
  http_response(socket,content=charToRaw(paste(body,"</body></html")))
}

w = create_server(webpage=httpd, port=9999)
cat("Direct your browser to http://localhost:9999\n")
while(TRUE) service(w)
```

5 Tips and miscellaneous notes

5.1 Binary data

Binary data is supported by IETF websocket protocol versions greater than 00. The websockets package supports the older 00 protocol with ASCII-only data, as well as binary data transfers with newer clients. At the date of this writing, the only commonly available web browser supporting the new protocols is Google Chrome (browser version 14 and greater), which uses the IETF version draft-ietf-hybi-thewebsocketprotocol-08.

JSON is probably a good non-binary choice to use when interacting with Javascript and the data size is not too large. The suggested RJSONIO package helps map many native R objects to JSON and vice versa, greatly facilitating interaction between R and Javascript.

5.2 Setting generic HTTP function handler callbacks

Assign a generic HTTP handler with set_callback function using the callback name "static" as in the following example:

```
hello = function(socket, header) {
  http_response(socket,content=charToRaw("HELLO"))
}
server = create_server(port=9999)
set_callback('static', hello, server)
```

Alternatively, use the webpage parameter in the create_server function.

5.3 Limitations

The websockets package does not yet automatically support the IETF WebSocket message fragmentation protocol. Users may implement the protocol manually, or limit the package use to single-frame messages.

The websockets package uses the R options interface to set the websockets_max_buffer_size variable to a default value of 16777216. Message frames exceeding this value will be truncated. Users may change the maximum buffer size value by resetting the option.

Extensions as specified by the IETF draft specification are not yet supported.

High-level control of the data framing headers are not yet exposed to users, but will be in a future package version.