it's about time

# Technical Whitepaper

## Kdb+ and WebSockets

**Author:**

Chris Scott works a kdb+ consultant at one of the world's largest financial institutions developing a range of kdb+ applications which use WebSockets as a form of communication.  Chris has also been involved in designing HTML5 training courses and building HTML5 mobile and desktop applications for First Derivatives' Kx technology.

## TABLE OF CONTENTS

# 1 INTRODUCTION

Since the release of kdb+ 3.0, it has been possible to make use of WebSockets when connecting to a kdb+ process. This has big implications on how we can build front-end applications that use a kdb+ back-end, in particular for applications wanting to display real-time information.

WebSockets are supported by most modern browsers, and so for the first time it is possible to hook a web application up directly to a kdb+ process to allow real-time communication over a persistent connection. This gives us the ability to build a pure HTML5 real-time web application which can connect to kdb+ through JavaScript, which is native to web browsers.

Using web applications to build GUIs for kdb+ applications is becoming increasingly popular. Only a web browser is required to run the GUI so it is OS independent. This means that when developing the GUI, only one codebase is required for all platforms and maintenance and upgrades are much easier and efficient to perform.

This whitepaper will introduce what WebSockets are and what benefits they hold over standard HTTP. It will also take the reader through the set-up of a simple web page that uses WebSockets to connect to a kdb+ process and the steps involved in passing data through the connection.

We will look at different methods for converting data between kdb+ and JavaScript and ways to help ensure the connection is secure.

The paper will finish up by providing a full working example (including code) of a simple web application which uses a kdb+ to provide real-time updating tables based on user queries.

**NOTE:** As well as q, this paper will make significant use of HTML, CSS and JavaScript. A basic understanding of these will be necessary to take full advantage of this paper, though any complex structures will be examined here in detail. For a crash course on the above languages, please refer to the following resources:

- http://www.w3schools.com
- http://www.codecademy.com

Throughout the paper, kdb+ code will be in grey text boxes while HTML and JavaScript code will be in green text boxes.

# 2 WEBSOCKETS

## 2.1 What Are WebSockets

WebSockets are a long awaited evolution in client and web server communication technology. They provide a protocol between a client and server which runs over a persistent TCP connection. The client-server connection can be kept open as long as needed and can be closed by either the client or the server. This open connection allows bi-directional, full-duplex messages to be sent over the single TCP socket connection - the connection allows data transfer in both directions, and both client and server can send messages simultaneously. All messages sent across a WebSocket connection are asynchronous.

Without WebSockets, bi-directional messaging can be forced by having distinct HTTP calls for both the client sending requests and the server publishing the responses and updates. This requires either the client to keep a mapping between outgoing and incoming messages, or the use of a proxy server in between the client and server (known as HTTP tunneling). WebSockets simplify this communication by providing a single connection which both client and server can send messages across.

WebSockets were designed to be implemented in web browsers and web servers, but they can be used by any client or server application. The ability for bi-directional real-time functionality means it provides a basis for creating real-time applications on both web and mobile platforms. The WebSocket API and Protocol have both been standardised by W3C and the IEFT respectively.

## 2.2 Why Not Just Use .z.ph?

It has previously been possible to connect to a kdb+ process using HTTP. HTTP requests could be processed using the `.z.ph` and `.z.pp` handlers. To illustrate this, simply start up a q process with an open port and then type the `hostname:port` into your web browser. This will give a very basic view of the q process.

Straight out of the box, this is very simple, and provides a useful interface for viewing data and running basic queries without being limited to the character limits of a q console. If, however, you want to do more than just simple analysis on the q process, this method presents a few drawbacks:

1) Customisation is quite a complicated process that requires you to manipulate the functions in the `.h` namespace which form the basis of the in-built HTTP server code. The HTML markup is generated entirely by the q process.
2) Data must be requested by the browser, so some form of polling must occur in order to update the webpage. This makes the viewing of real-time data impossible without the continuous polling of the kdb+ server.
3) `.z.ph` uses synchronous messaging, and the webpage is effectively refreshed every time a query is sent.

Instead of relying on `.z.ph` to serve the entire web page, an application could make use of AJAX techniques to send asynchronous GET and POST requests to a kdb+ server. `.z.ph` and `.z.pp` can be modified to handle these requests. AJAX (Asynchronous JavaScript And XML) is a descriptive name that covers a range of web development techniques that can be used to provide asynchronous messaging between a web application and server.

This allows requests to be made by the client in the background without having to reload the web page, and as there is no reliance on `.z.ph` to generate the HTML markup, the application can be fully customised independent of the kdb+ process. In order to receive updates the application will still have to poll the server, which is not ideal for real-time data. The full implementation of this method is beyond the scope of this whitepaper.

As WebSocket connections are persistent, the server is able to push updates to the client rather than relying on the client to poll for the information. Additionally, while each HTTP request and response must include a TCP header, WebSockets only require this header to be sent during the initial handshake.

# 3 CONNECTING TO kdb+ USING WEBSOCKETS

## 3.1 The Handshake

In order to initialise a WebSocket connection, a WebSocket 'handshake' must be successfully made between the client and server processes. First, the client sends a HTTP request to the server to upgrade from the HTTP protocol to the WebSocket protocol:

```
// Client WebSocket request header
GET /text HTTP/1.1
Host: localhost:5001
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xc0DPEL132mlMtGdbWJPGQ==
Sec-WebSocket-Version: 13
```

The server then sends a HTTP response indicating that the protocol upgrade was successful:

```
// Server WebSocket response header
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1thgMROs9ylOWOMkc2WUWGRzWdY=
```

## 3.2 .z.ws Message Handler

Like the `.z.ph` HTTP GET handler, kdb+ has a separate message handler for WebSockets called `.z.ws`, meaning all incoming WebSocket messages will be processed by this function. There is no default definition of `.z.ws`; it should be customised by the developer to handle the incoming messages as required. Later, several examples of a customised `.z.ws` handler will be shown, but initially we will look at a very basic definition:

```
q) .z.ws:{neg[.z.w].Q.s value x;}
```

For now, let's assume that `x` (the query from the client) is being passed in as a string. `value x` simply evaluates the query, and passing this into `.Q.s` will present the result in console output format (i.e. plain text) .

As mentioned before, all messages passed through a WebSocket connection are asynchronous. This means that we must handle the server response within the `.z.ws` message handler. `neg[.z.w]` does this by asynchronously pushing the results back to the handle which raised the request.

From the server side, using `neg[.z.w]` within the handler will push the result back to the client once it has been evaluated. The client does not wait for a response and instead the open WebSocket connection will receive the response some time later, so we must handle this on the client side as well.

Fortunately, JavaScript has a native WebSocket API which allows us to handle this relatively easily. The main parts of the JavaScript API are explained below.

```
<script>
function connect(){
  if ("WebSocket" in window){  // check if WebSockets supported
     // open a WebSocket
     var ws = new WebSocket("ws://host:port");
     ws.onopen = function(){
        // called upon successful WebSocket connection
     };
     ws.onmessage = function(msg){
        // called when the client receives a message
     };
     ws.onclose = function(){
        // called when WebSocket is closed.
     };
  }
  else{
     // the browser doesn't support WebSockets
  }
}
ws.send(msg)    // function to handle sending a message
</script>
```

### 3.3 Simple Example

Here is an example of a simple client application connecting to a kdb+ server using a WebSocket. This is a slightly modified version of the example that can be found on the code.kx.com WebSocket cookbook. http://code.kx.com/wiki/Cookbook/Websocket

First, start up our kdb+ server and set our `.z.ws` handler. Here we will add some error trapping to the handler to send the error message to the client.

```
C:\Users\Chris>q -p 5001  // Open a port on 5001
KDB+ 3.2 2014.09.21 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q) .z.ws:{neg[.z.w].Q.s @[value;x;{`$x}];}
q)
```

Next we create a HTML document we will use to connect to the above kdb+ server. The below code can be copied into a text editor and saved as a .html file, then opened in any modern web browser.

```
<!doctype html>
<html>
<head>
      <title>WebSocket Demo</title>
</head>
<!-- when page loads, call connect function -->
<body onload="connect();">
  <!-- The application will consist of a text box to enter queries, a
button to execute the queries, and an area to display the results -->
  <textarea id="txtInput" placeholder="q)"></textarea>
  <!-- the onclick event handler will make a call to JavaScript -->
  <button id="cmdInput" onclick="send();">Go</button>
  <div id="txtOutput"></div>
  <script> // JavaScript code goes inside 'script' tags
  var ws, cmd = "";  // initialise the JS variables required
  var input=document.getElementById("txtInput");
  var output=document.getElementById("txtOutput");
  function connect(){
  // this is where we define the WebSocket API functionality
    if ("WebSocket" in window) {
      ws = new WebSocket("ws://localhost:5001/");
  // show the state of the WebSocket connection in the display area
      output.value="connecting...";
      ws.onopen=function(e){output.innerHTML="connected";}
      ws.onclose=function(e){output.innerHTML="disconnected";}
      ws.onmessage=function(e){
      // when a message is received, print the message to the display area
along with the input command
        output.innerHTML = cmd +
             e.data.replace(/ /g, ' ').replace(/\n/g, '<br />') +
output.innerHTML;
      // the message is in plain text, so we need to convert ' ' to
' ' and '\n' to '<br />' in order to display spaces and newlines
correctly
        cmd="";
      }
      ws.onerror=function(e){out.value=e.data;}
    } else alert("WebSockets not supported on your browser.");
  }
  function send(){
   // store the input command as tmpInput so that we can access it later
to print in with the response
    cmd = "q)" + input.value + "<br />";
   // send the input command across the WebSocket connection
    ws.send(input.value);
   // reset the input test box to empty, and focus the cursor back on it
ready for the next input
    input.value=""; input.focus();
   }
  </script>
  <style>  /* define some CSS styling on page elements */
  #txtInput {width: 85%; height: 60px; float:left; padding: 10px;}
  #cmdInput {margin-left: 10px; width:10%; height:80px; font-weight:
bold;}
```

```
  #txtOutput {width: 96%; height: 300px; font-family: courier new;
padding: 10px; border: 1px solid gray; margin-top: 10px; overflow:auto;}
  </style>
</body>
</html>
```

This example sets our webpage up as a q console. As it is using `.Q.s` to convert the kdb+ output into string format, it is subject to the restrictions of the q console size set by `\c`.

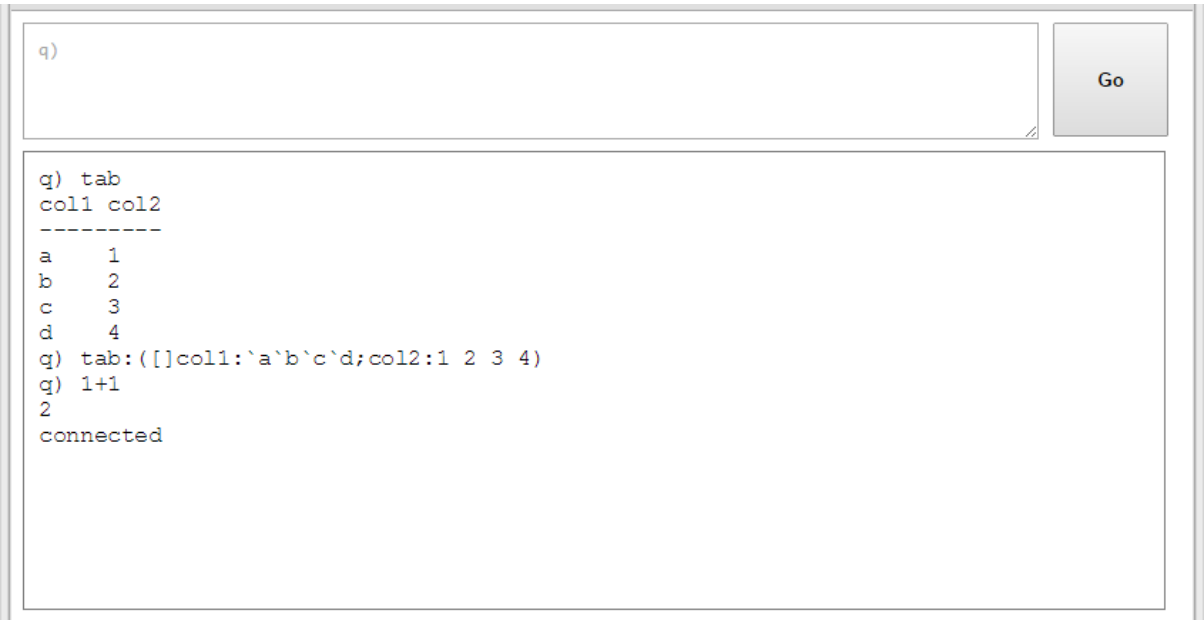



**Figure 3.2 – This example provides a web console application which allows q commands to be executed from the browser**

### 3.4 Pushing Data To The Client Using neg[h]

Above, we have used `neg[.z.w]` within the `.z.ws` handler to return the result to the client immediately after it is received and evaluated. But in some situations we don't want to just provide a response, but rather set up a subscription that continually pushes data through our WebSocket connection.

As the WebSocket is persistent and obeys normal IPC protocol, we can push data through it asynchronously at any time using `neg[h]`, where `h` is the WebSocket handle. To see this, in the input box of the example application above, type the following code and click 'Go'.

```
.z.ts:{[x;y]neg[x].Q.s .z.T}[.z.w]; system"t 1000"
```

You should now see the current time being output every second, without the browser having to poll for the data.

## 4 CONVERTING kdb+ TO JSON

Converting kdb+ into string format using `.Q.s` means that not only is the message limited by the console size, but also that as the client is receiving plain text the data will be unstructured and very difficult to parse into something that can be manipulated by the client.

JavaScript has a built-in technology called JSON (JavaScript Object Notation) which can be used to create JavaScript objects in memory to store data. These objects are collections of name/value pairs which can be easily accessed by JavaScript and used to build the data structures displayed on the web page.

We want to make use of JSON to hold our data so that we can easily display and manipulate our data on the client. There is a very important relationship between kdb+ and JSON – kdb+ dictionaries and JSON objects are comparable. This means that we can parse our kdb+ data into JSON structures very easily, with tables just becoming arrays of JSON objects.

Depending on the application, it may be of benefit to do this conversion on the server side within the q process, or on the client side in the JavaScript code. We will explore the two options along with an example to show both in action.

### 4.1 Using json.k Within a q Process (Server-side Parsing)

kdb+ 3.2 includes a new `.j` namespace which contains functions for translating between kdb+ and JSON format. For older versions of kdb+, Kx provides the same functionality inside a script named json.k which can simply be loaded into a q process. That script can be found here: http://kx.com/q/e/json.k

There are two main functions that we will make use of:
- **`.j.j`** parses kdb+ into a JSON string
- **`.j.k`** parses a JSON string into kdb+

```
C:\Users\Chris\Documents\Whitepaper>q
KDB+ 3.2 2014.09.21 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q)tab:([]col1:`a`b`c`d;col2:1 2 3 4)
q)tab
col1 col2
---------
a    1
b    2
c    3
d    4
q).j.j tab
"[{\"col1\":\"a\",\"col2\":1},{\"col1\":\"b\",\"col2\":2},
{\"col1\":\"c\",\"col2\":3},{\"col1\":\"d\",\"col2\":4}]"
q).j.k "{\"a\":1,\"b\":2,\"c\":3,\"d\":4}"
a| 1
b| 2
c| 3
d| 4
```

Here is an example of how we can use this to manipulate data in our web app. First set up our server and use `.j.j` to convert the kdb+ output to JSON:

```
C:\Users\Chris>q -p 5001  // Open a port on 5001
KDB+ 3.1 2014.03.27 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q) .z.ws:{neg[.z.w].j.j @[value;x;{`$"'",x}];}
q)
```

We can edit the `ws.onmessage` JavaScript function from the example in Section 2 to now handle the messages coming through as JSON strings rather than just plain text.

```
ws.onmessage=function(e){
      var t,d = JSON.parse(e.data);
// we will use the variable t to build our HTML
// we parse the JSON string into a JSON object using JSON.parse
      if (typeof d == "object") { // either a table or dictionary
            if (d.length) {  // if d has a length it is a table
// we will to iterate through the object wrapping it in the HTML table
tags
               t = '<table border="1"><tr>'
               for (var x in d[0]) {
// loop through the keys to create the table header
                  t += '<th>' + x + '</th>';
               }
               t += '</tr>';
               for (var i = 0; i < d.length; i++) {
// loop through the rows, putting tags around each column value
                  t += '<tr>';
                  for (var x in d[0]) {
                     t += '<td>' + d[i][x] + '</td>';
                  }
                  t += '</tr>';
               }
               t += '</table>';
            } else {
// if the object has no length, it must be a dictionary, we will iterate
over the keys to print the key|value  pairs as would be displayed in a q
console
                  t = "";
                  for (var x in d) {
                        t += x + " | " + d[x] + "<br />";
                  }
            }
      } else {
// if not an object, then the message must have a simple data structure,
in which case we will just print it out.
            t = d;
      };
       output.innerHTML = cmd + t + "<br />" + output.innerHTML;
 }
```

This gives us the same web console application as before, but the distinction is that we are now handling JSON objects which are easier to manipulate than strings of plain text. The JavaScript code will also be aware of data type information which is not the case with plain strings.

Using the `.j` functions within the q process is very straight forward, and if it is suitable for the application to transfer messages across the WebSocket as strings this can be a good solution. However, in some cases it may be preferable to serialise the data into binary format before transmitting it across the connection.

### 4.2 Using c.js Within JavaScript (Client-side Parsing)

Instead of parsing the data within the kdb+ server, we could instead use `-8!` to serialise the data into kdb+ binary form and then deserialise it on the client side directly into a JSON object. With client machines generally being a lot more powerful than in the past, it is reasonable to provide the client side JavaScript code with some extra workload.

This approach requires a little more understanding of JavaScript. However, Kx provides the script c.js which contains the functionality to serialise and deserialise data on the client side. The `deserialize` function converts kdb+ binary data into JSON, while the `serialize` function will convert our message into kdb+ binary format before sending it to the server. c.js can be found here: http://kx.com/q/c/c.js,

Section 4.1 showed how we can parse q structures into JSON strings and send them to our client. In this example, we will instead do all of the parsing on the client side to produce the same result. The client will send strings to the server but the server will send serialised data back to the client. Outgoing messages are serialised using the `-8!` operator:

```
C:\Users\Chris>q -p 5001  // Open a port on 5001
KDB+ 3.2 2014.09.21 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q) .z.ws:{neg[.z.w] -8! @[value;x;{`$"'",x}];}
q)
```

We need to make a few adjustments to our JavaScript code from section 4.1 in order to handle the serialised responses. Firstly, we need to load the c.js script into our web page.

```
<script src="http://kx.com/q/c/c.js"></script>
```

The above line should be placed on the line before the opening `<script>` tags which surround the rest of the JavaScript code.

When we define our WebSocket connection, we need to tell the JavaScript code to expect the messages to be coming through in binary. For this we use `ws.binaryType = 'arraybuffer'`. Place this line on the line after we have set up the new WebSocket connection.

```
ws = new WebSocket("ws://localhost:5001/");
ws.binaryType = 'arraybuffer';
```

Next, we need to edit the `ws.onmessage` function. As we are deserialising the message straight into JSON, we do not need to use the `JSON.parse` function. Instead, we simply replace it with the `deserialize` function provided in c.js.

```
var t,d = deserialize(e.data);
```

The rest of `ws.onmessage` is identical to the example in Section 4.1.

More information on serialisation can be found here:
http://code.kx.com/wiki/Reference/ipcprotocol

## 5 WEBSOCKET SECURITY

Security is one of the biggest concerns in any system. It is of the utmost importance that, especially when dealing with sensitive financial information, users only have access to view and edit data they are allowed to see.

The methods used to protect kdb+ processes can be extended to cover WebSockets. There is one caveat in terms of username/password authentication which we will discuss below. A full discussion on kdb+ permissioning is beyond the scope of this whitepaper; instead, please refer to the July 2013 whitepaper in the q for Gods series:
http://www.firstderivatives.com/lecture_series_pp.asp?downloadflyer=q_for_Gods_July_2013

### 5.1 Username and Password

Client authentication is not defined in the WebSocket protocol, and in most cases it is up to the developer to implement an authentication method of their choice. This means that being prompted for a username and password by the browser when the WebSocket makes a request cannot be guaranteed. Firefox does support this, but Chrome and other browsers do not, as the following example will show.

First create our user/password file and start the server using the –u argument.

```
C:\Users\Chris>type users.txt
chris:password
user2:password2
C:\Users\Chris>q -p 5001 -u users.txt // Open a port on 5001
KDB+ 3.2 2014.09.21 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q) .z.ws:{neg[.z.w] -8! @[value;-9!x;{`$"'",x}];}
```

If we try to connect to the process in Firefox it will prompt for a username and password and, upon correct entry, successfully create the connection. Chrome, on the other hand, will not ask for authentication details and so the connection will not be successful. When the connection is not successful due to authentication, a response code of 401 (unauthorized) will be returned to the client.

Behind the scenes, Firefox is actually sending two requests. When it receives the first 401 response code, it prompts for a username and password and then sends a new WebSocket request with an additional 'Authorization' option in the header. Chrome and other browsers do not currently do this for WebSocket connections.

As a workaround, we can make a standard HTTP request before we initalise the WebSocket. Once authorization has been granted to the session, it will continue to include the Authorization option in any subsequent HTTP headers that are sent. When we then try to initalise our WebSocket connection, the header will include this information and access will be granted.

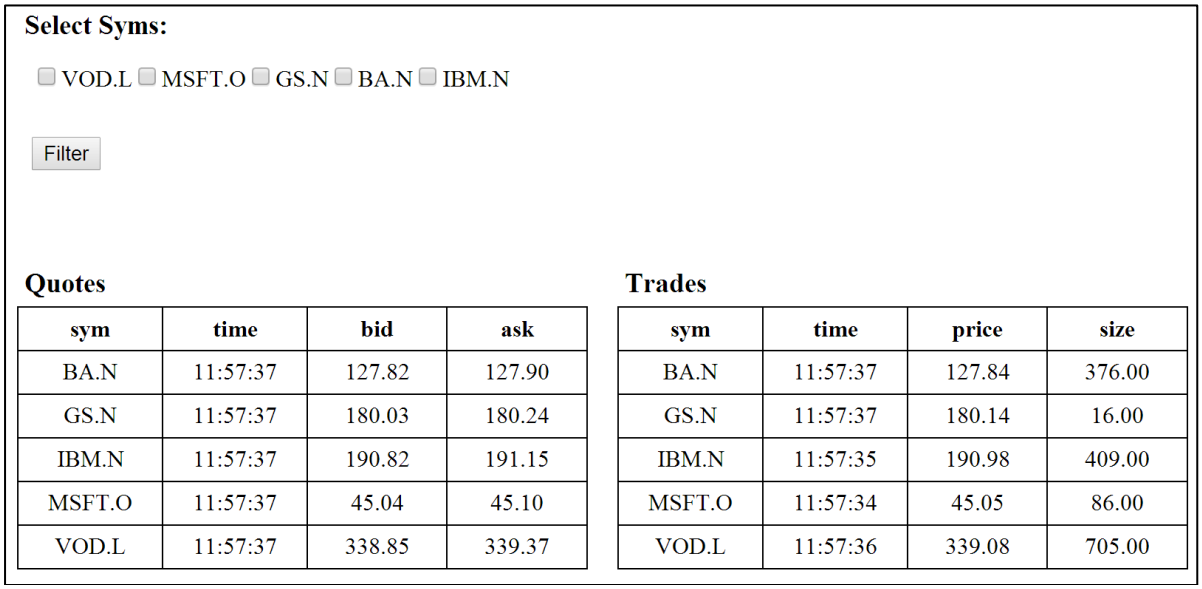
As mentioned in Section 3.2, all WebSocket messages will be processed by the `.z.ws` handler. To fully secure a kdb+ system, this handler should be locked down in the same manner as the `.z.pg`, `.z.ps`, `.z.ph` and `.z.pp` handlers.

**NOTE**: Stunnel can be used to secure any WebSocket server using the OpenSSL library. For more information on Stunnel, see:
http://code.kx.com/wiki/Cookbook/Websocket#Secure_sockets:_stunnel

## 6 A SIMPLE EXAMPLE – REAL-TIME DATA

This section will present a simple example in which some tables will be updated in the browser in real-time.

The full code can be found in Appendix A along with start-up instructions in order to get the example working. Some of the key features will be explained here. Once the example is successfully running you should be able to see the following tables in your browser continuously updating:




| sym | time | bid | ask |
|---|---|---|---|
| BA.N | 11:57:37 | 127.82 | 127.90 |
| GS.N | 11:57:37 | 180.03 | 180.24 |
| IBM.N | 11:57:37 | 190.82 | 191.15 |
| MSFT.O | 11:57:37 | 45.04 | 45.10 |
| VOD.L | 11:57:37 | 338.85 | 339.37 |

| sym | time | price | size |
|---|---|---|---|
| BA.N | 11:57:37 | 127.84 | 376.00 |
| GS.N | 11:57:37 | 180.14 | 16.00 |
| IBM.N | 11:57:35 | 190.98 | 409.00 |
| MSFT.O | 11:57:34 | 45.05 | 86.00 |
| VOD.L | 11:57:36 | 339.08 | 705.00 |

**Figure 6.1 – The web page shows the last quote and trade values for each symbol, along with giving the user the ability to filter the syms in view**

The idea behind the pubsub mechanism here is that a client will make subscriptions to specific functions and provide parameters that they should be executed with. Standard kdb+ IPC allows messages to be sent as a list, where the first element is a function name and the following elements are function parameters. We will mimic that functionality here. Messages will be sent as lists (arrays in JavaScript) in the format `(function;arg1;arg2;…)`.

```
.z.ws:{value -9!x}
```

In this example we will be serializing all messages that are sent from the client to the server. Accordingly, on the server we will need to use `-9!` to deserialise the message into kdb+ format.

Next we initialize the `trade` and `quote` tables and `upd` function to mimic a simple Real Time Subscriber, along with a further table called `subs` which we will use to keep track of subscriptions.

```
// subs table to keep track of current subscriptions
subs:2!flip `handle`func`params`curData!"is**"$\:()
```

The `subs` table will store the handle, function name and function parameters for each client. As we only want to send updates to a subscriber when something has changed, we store the current data held by each subscriber so that we can compare against it later.

```
// pubsub functions
sub:{`subs upsert (.z.w;x;y;res:eval(x;enlist y));(x;res)}
pub:{neg[x] -8!y}
pubsub:{pub[.z.w] eval(sub[x];enlist y)}
.z.pc:{delete from `subs where handle=x}
```

The `sub` function will handle new subscriptions by upserting the handle, function name and function parameters into the `subs` table. When a new subscription is made we execute the function with the specified parameters, store the result in the `subs` table along with the other subscription details, and then return a list of `(functionName; data)`.

When we publish data to a client, we want to send it asynchronously and serialized. This is handled by the `pub` function. `pubsub` combines these two functions such that the `subs` table is updated with the latest data and this data is then pushed to the client. Shortly we will see how the `subs` table can be used to continually push updates to the client.

Very importantly, if a connection is closed we want to make sure that there are no subscriptions left for that handle in the subs table. This is handled in `.z.pc`.

The functions that can be called and subscribed to by clients through the WebSocket should be defined as necessary. In this example, we have defined some simple functions to display the last record for each `sym` in both the `trade` and `quote` tables.

Now that we have defined the subscription and publication methods, we need a way of determining when data has changed and then pushing the latest information to the client. The `refresh` function will run on a timer and execute each function in the subs table with the specified parameters. If any of the function calls produce a different output to what the client holds (stored in the `curData` column) then the latest data will be pushed to that client.

```
refresh:{
  update curData:{[h;f;p;c]
    if[not c~d:eval(f;enlist p);pub[h] (f;d)]; d
  }'[handle;func;params;curData] from `subs
 }
```

One thing to note is that this example keeps the amount of data sent to the client very small, so there is no problem with storing a copy of what each client currently has for each subscription.

In the JavaScript code for the client, most of the functionality has been seen in previous examples. One thing to highlight is how we can use the messages as arrays to know which JavaScript function to call.

```
ws.onmessage=function(e){
   var d =  deserialize(e.data);
   window[d.shift()](d[0]);
};
```

Once the message in `e.data` has been deserialised, it will be in the format `['functionName',parameters]`. This keeps a certain symmetry between how the messages are handled on both client and server. In order to execute a function when we have the name as a string, we can use `window['functionName']`. `d.shift()` returns the first element of the array `d`, whilst also persisting `d` with the first element removed.

Inside the `tableBuilder` function, there is some formatting for times and numbers.

```
data[i][x].toLocaleTimeString().slice(0,-3)
data[i][x].toFixed(2)
```

The end result is a simplistic, interactive, real-time web application showing the latest trade and quote data for a range of symbols. Its intention is to help readers understand the basic concepts of kdb+ and WebSocket integration.

## 7 CONCLUSION

This whitepaper has shown how WebSockets can be used as part of an HTML5 GUI to connect to a q process to allow persistent, real-time communication between kdb+ and a web browser.

There are different methods to sending messages over the connection, and we have seen how to parse kdb+ data into JSON objects using both server-side and client-side methods. Both of these methods have their benefits and drawbacks so it is important to consider the application infrastructure when deciding which method will be most suitable. Serialising data across the connection is easy to achieve using the `-8!` and `-9!` functions on the kdb+ server and the c.js code provided by Kx on the JavaScript client.

Access control is a crucial aspect of any system, and we have seen that although basic user authentication is not part of the standard WebSocket protocol it can be overcome using techniques which are already employed in front-end applications using AJAX as a method of communication.

Delta Dashboards, part of the Delta Suite from First Derivatives, provides a range of great ways to visualise and analyse both kdb+ data and data from other sources while leveraging the enterprise functionality of Delta for authentication/authorisation. For more information on Delta Dashboards please visit here.

All tests were run using kdb+ version 3.2 (2014.09.21)

*The release of kdb+ 3.2 contained updates to allow kdb+ processes to be used as a WebSocket client, and also for compression to be used over a WebSocket connection. For more information: http://code.kx.com/wiki/Cookbook/Websocket#Compression*

## APPENDIX A

Below are four separate scripts which can be copied into a text editor and saved with the respective names in the same directory. Start the q processes up first, and then open the HTML file in a web browser.

**pubsub.q**
Start this process first. It will create the q interface for the WebSocket connections and contains a simple pubsub mechanism to push data to clients when there are updates.

```
// q pubsub.q -p 5001

.z.ws:{value -9!x}

// table and upd definitions
trade:flip `time`sym`price`size!"nsfi"$\:()
quote:flip `time`sym`bid`ask!"nsff"$\:()
upd:insert

// subs table to keep track of current subscriptions
subs:2!flip `handle`func`params`curData!"is**"$\:()

// pubsub functions
sub:{`subs upsert (.z.w;x;y;res:eval(x;enlist y));(x;res)}
pub:{neg[x] -8!y}
pubsub:{pub[.z.w] eval(sub[x];enlist y)}
.z.pc: {delete from `subs where handle=x}

// functions to be called through WebSocket
loadPage:{pubsub[;`$x]each `getSyms`getQuotes`getTrades}
filterSyms:{pubsub[;`$x]each `getQuotes`getTrades}

// get data methods
getData:{
  w:$[all all null y;();enlist(in;`sym;enlist y)];
  0!?[x;w;enlist[`sym]!enlist`sym;()]
 }
getQuotes:{getData[`quote] x}
getTrades:{getData[`trade] x}
getSyms:{distinct (quote`sym),trade`sym}

// refresh function - publishes data if changes exist, and updates subs
refresh:{
  update curData:{[h;f;p;c]
    if[not c~d:eval(f;enlist p);pub[h] (f;d)]; d
  }'[handle;func;params;curData] from `subs
 }
// trigger refresh every 100ms
.z.ts:{refresh[]}
\t 100
```

**fh.q**

This will generate dummy trade and quote data and push it to the pubsub process. The script can be edited to change the number of symbols and frequency of updates.

```
// q fh.q
h:neg hopen `:localhost:5001 // connect to rdb
syms:`MSFT.O`IBM.N`GS.N`BA.N`VOD.L // stocks
prices:syms!45.15 191.10 178.50 128.04 341.30 // starting prices
n:2 // number of rows per update
flag:1 // generate 10% of updates for trade and 90% for quote
getmovement:{[s] rand[0.001]*prices[s]} // get a random price movement
// generate trade price
getprice:{[s] prices[s]+:rand[1 -1]*getmovement[s]; prices[s]}
getbid:{[s] prices[s]-getmovement[s]} // generate bid price
getask:{[s] prices[s]+getmovement[s]} // generate ask price
// timer function
.z.ts:{
 s:n?syms;
 $[0<flag mod 10;
 h(`upd;`quote;(n#.z.N;s;getbid'[s];getask'[s]));
 h(`upd;`trade;(n#.z.N;s;getprice'[s];n?1000))
 ];
 flag+:1;
 };
// trigger timer every 100ms
\t 100
```

*This Feed Handler code is courtesy of the August 2014 whitepaper in the q for Gods series:*
*http://www.firstderivatives.com/Products_pdf.asp?downloadflyer=q_for_Gods_Aug_2014*

**websockets.html**

Due to the length of code required for this example, the JavaScript and HTML code have been split into separate files

```
<!doctype html>
<html>
<head>
   <title>WebSocket PubSub Example</title>
</head>
<body onload="connect();">
   <!-- Create a section to filter on syms -->
   <section class="select">
      <h3>Select Syms: </h3>
      <div id="selectSyms"></div>
      <button type="submit" onclick="filterSyms();">Filter</button>
   </section>
   <!-- Set up placeholders to display the trade and quote outputs -->
   <section id="quotes" class="display">
      <div class="split">
         <h3>Quotes</h3>
         <table id="tblQuote"></table>
      </div>
      <div class="split">
         <h3>Trades</h3>
         <table id="tblTrade"></table>
      </div>
   </section>
   <!-- Load JavaScript files -->
   <script src="http://kx.com/q/c/c.js"></script>
   <script src="websockets.js"></script>

   <style>  /* define some CSS styling on page elements */
      section {margin:10px;padding:20px;width:95%;}
      button {margin:10px;}
      h3 {margin:5px;}
      table {border-collapse:collapse;text-align:center;width:100%;}
      td,th {border:1px solid black;padding:5px 20px;width:25%}
      .split {float:left;width:45%;margin-right:20px;display:table;}
      #selectSyms {padding:10px;min-height:30px;}
   </style>
</body>
</html>
```

**websockets.js**

This script will be loaded into the web page by the HTML. Make sure this is saved as a .js file in the same directory as the above HTML file.

```
// initialise variable
var ws,syms = document.getElementById("selectSyms"),
     quotes = document.getElementById("tblQuote"),
     trades = document.getElementById("tblTrade");
function connect(){
   if ("WebSocket" in window) {
      ws = new WebSocket("ws://localhost:5001");
      ws.binaryType = 'arraybuffer'; // using serialisation
      ws.onopen=function(e){
   // on successful connection, we want to create an initial subscription
to load all the data into the page
         ws.send(serialize(['loadPage',[]]));
      };
      ws.onclose=function(e){console.log("disconnected");};
      ws.onmessage=function(e){
   // deserialise incoming messages
         var d =  deserialize(e.data);
   // messages should have format [‘function’,params]
   // call the function name with the parameters
         window[d.shift()](d[0]);
      };
      ws.onerror=function(e){console.log(e.data);};
   } else alert("WebSockets not supported on your browser.");
}
function filterSyms() {
  // get the values of checkboxes that are ticked and convert into an
array of strings
   var t = [], s = syms.children;
   for (var i = 0; i < s.length ; i++) {
      if (s[i].checked) { t.push(s[i].value); };
   };
  // call the filterSyms function over the WebSocket
   ws.send(serialize(['filterSyms',t]));
}
function getSyms(data) {
  // parse an array of strings into checkboxes
   syms.innerHTML = '';
   for (var i = 0 ; i<data.length ; i++) {
      syms.innerHTML += '<input type="checkbox" name="sym" value="' +
data[i] + '">' + data[i] + '</input>';
   };
}
function getQuotes(data) { quotes.innerHTML = tableBuilder(data); }
function getTrades(data) { trades.innerHTML = tableBuilder(data); }
function tableBuilder(data) {
  // parse array of objects into HTML table
   var t = '<tr>'
   for (var x in data[0]) {
      t += '<th>' + x + '</th>';
   }
```

```
   t += '</tr>';
   for (var i = 0; i < data.length; i++) {
      t += '<tr>';
      for (var x in data[0]) {
         t += '<td>' + (("time" === x) ?
data[i][x].toLocaleTimeString().slice(0,-3) : ("number" == typeof
data[i][x]) ? data[i][x].toFixed(2) : data[i][x]) + '</td>';
      }
      t += '</tr>';
   }
   return t ;
}
```