How to Give Your Sencha App Real-time Web Performance

James Schreuder

SenchaCon 2016

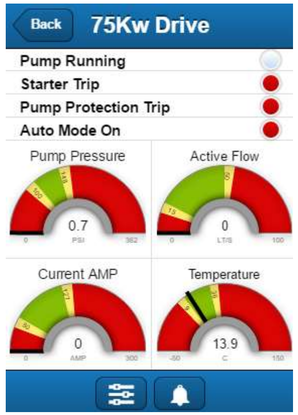
SCHREUDER ENGINEERING
SOFTWARE DESIGN AND ENGINEERING

James Schreuder

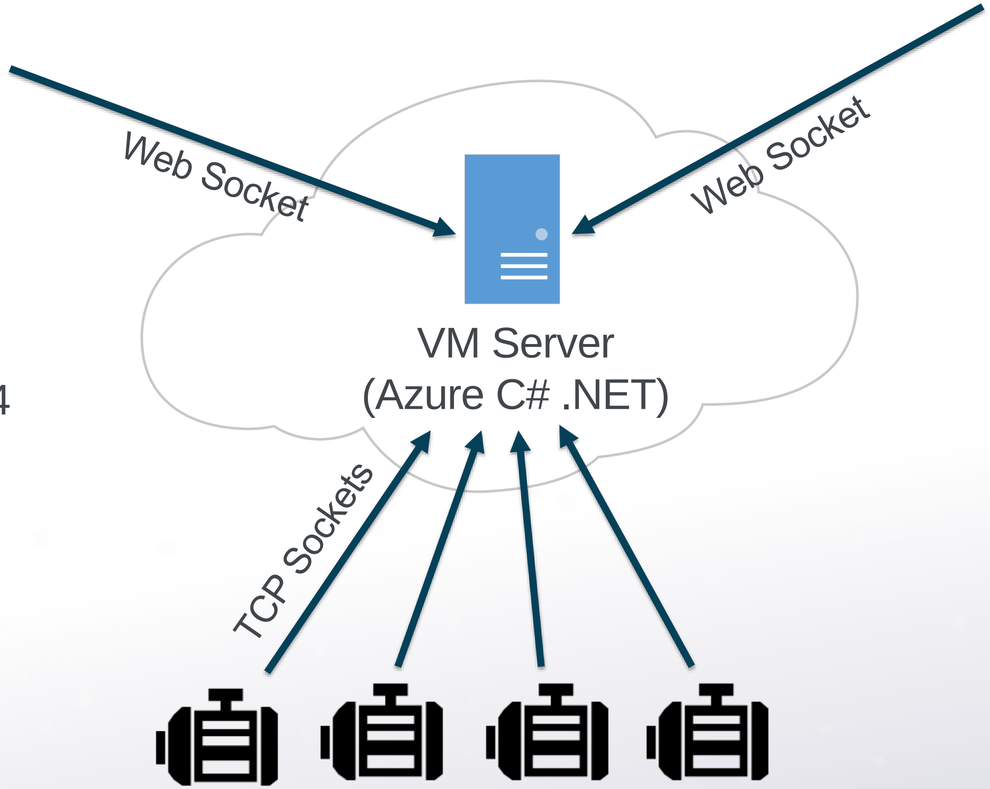
Principal Software Engineer

www.schreuder.com.au

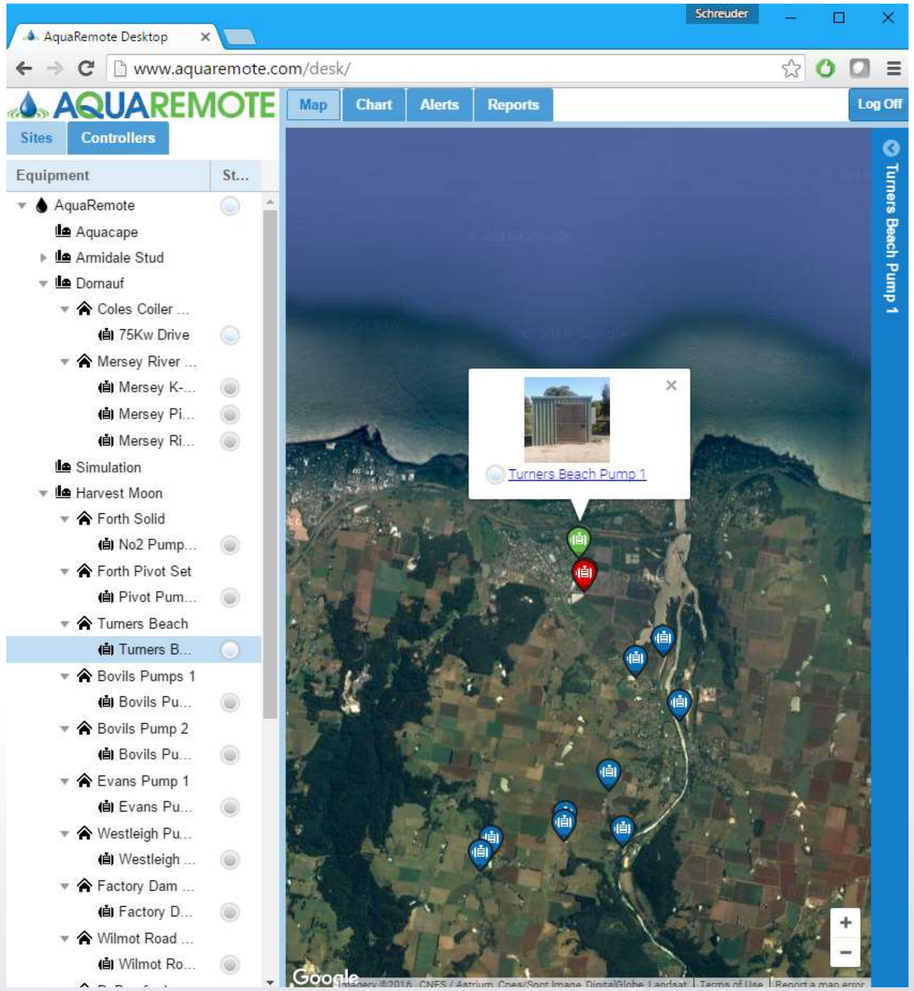
AQUAREMOTE



Sencha Touch 2.4
(Cordova)



Pump Controllers
(Embedded C++/Linux)



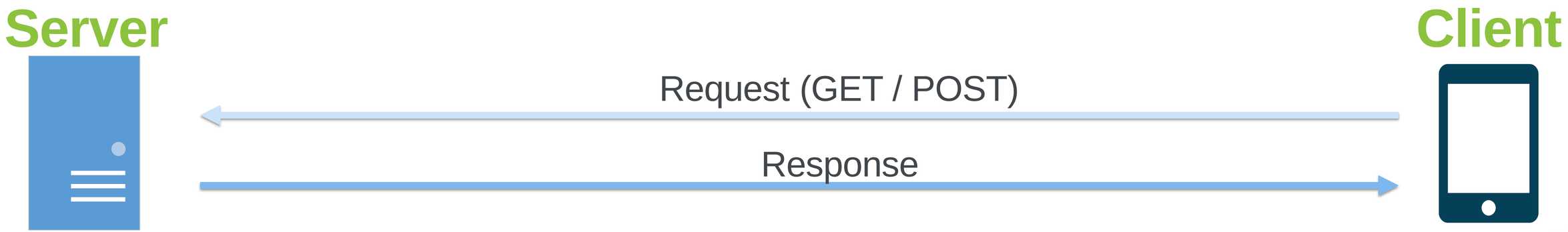
Web Application
(ExtJS 5)

Agenda

- Part 1: Data Transfer over the Web
- Part 2: Real-time Data Transfer over the Web
- Part 3: Real-time Web Frameworks and Sencha
- Part 4: Real-time Web over Wireless Networks

Part 1:
Data Transfer over the Web

HTTP 1.0/1.1



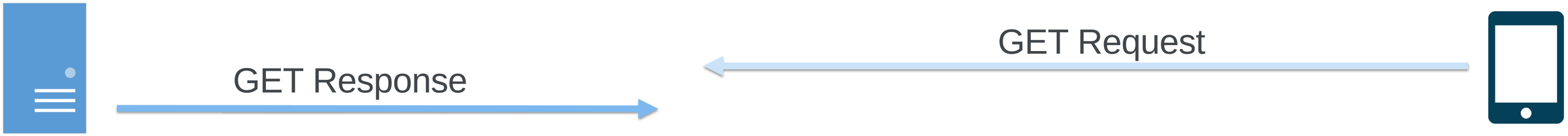
- Each HTTP request results in a new TCP socket connection
- Request sent, processed and the response returned within same TCP socket
- Half-duplex communication
- Requests to Web Server are stateless
- Client is required to initiate communication with the Web Server

HTTP Polling: Non Real-time


```
startHttpPolling(url, interval, request) {
        setInterval(function(){
            Ext.Ajax.request({
                    url: url,
                     method: 'POST',
                     params: { request: request },
                success: function(response, opts) {
                    var dataUpdate = response.responseText;
                    updateGui(dataUpdate);
                },
                failure: function(response, opts) {
                        // error!
                }
                dataType: "json"});
        }, interval);
    },
```

HTTP 1.1 Request/Response Example

Server **Client**




```
HTTP/1.1 200 OK
X-Powered-By: PHP/5.6.21
Content-Type: text/html; charset=UTF-8
Date: Tue, 16 Aug 2016 01:12:52 GMT
Accept-Ranges: bytes
Server: LiteSpeed
Connection: Keep-Alive
Content-length: 6286
```

```
GET / HTTP/1.1
Host: www.schreuder.com.au
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
        (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch Accept-Language: en-US,en;q=0.8
```

- Header = 200+ bytes
- *Not counting the body*

- Header = 360+ bytes

HTTP Long Polling: Near Real-time


```
startHttpLongPoll: function(url, request, timeout) {
        var ajaxId = Ext.Ajax.request({
            url: url,
            method: 'POST',
            params: { request : request },
            success: function(xhr, opts) {
                    var dataUpdate = response.responseText;
                    updateGui(dataUpdate);
                    startHttpLongPoll(url, request, timeout);
            },
            failure: function(response, opts){
                // error!
            },
            useDefaultXhrHeader : false,
            timeout: timeout
        });
    },
```

HTTP Polling: Disadvantages

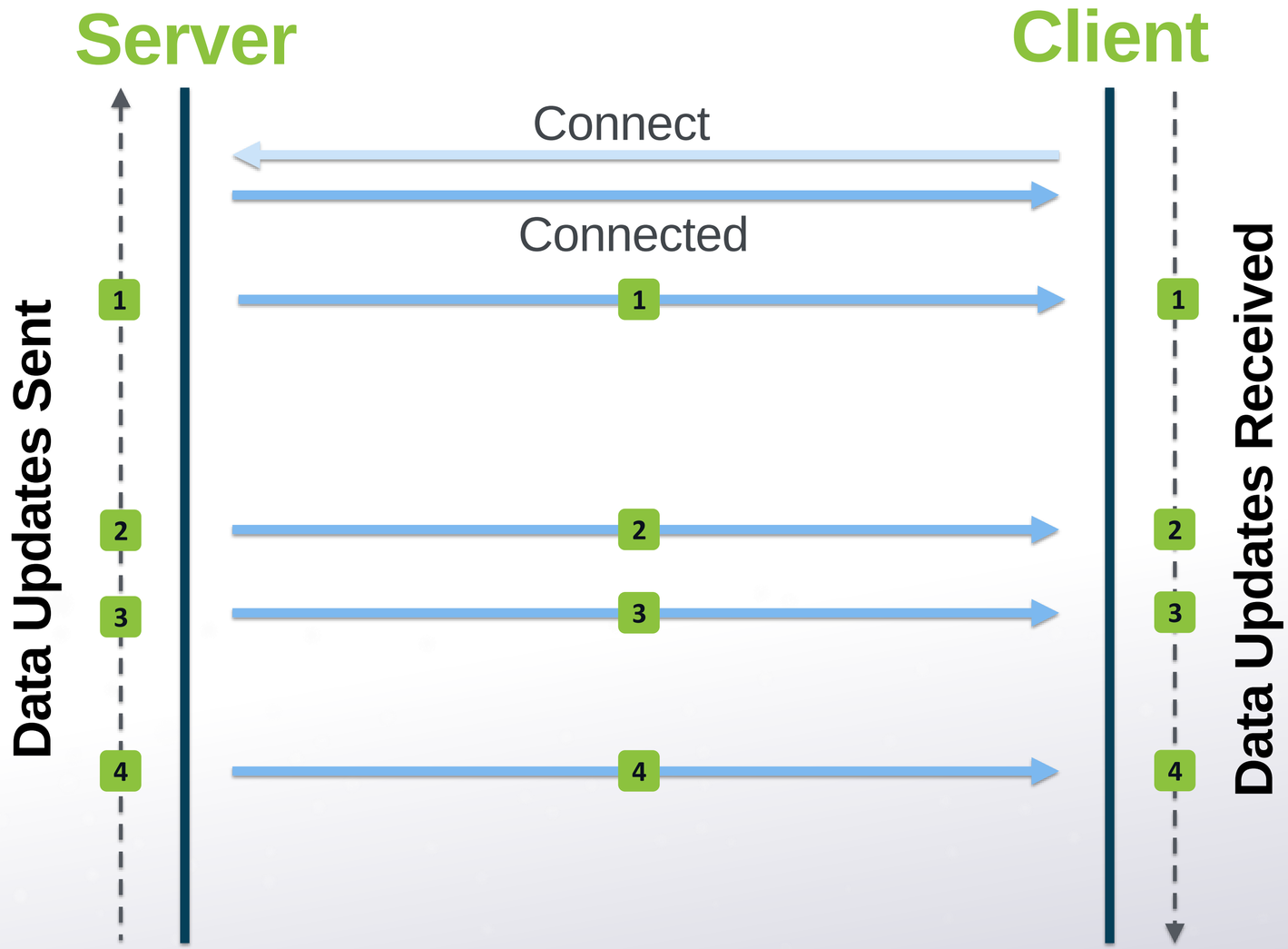
- HTTP data transfer is inherently half-duplex
- Large data overhead in HTTP Headers
- HTTP Polling:
  - New data updates are only available when polled
  - Need to balance polling frequency versus the data update arrival
- HTTP Long Polling:
  - Better than HTTP Polling as data updates are polled when they become available
  - New Long Poll request generated as each update arrives

Part 2:
Real-time Data Transfer over the Web

Web Sockets (RFC6455 – December 2011)

- TCP Socket initiated from client browser
- Minimal connection handshake
- Small message header overhead (between 6 and 14 bytes)
- Full-duplex and Bi-directional
  - Following connection handshake, either client or server can initiate communication
  - Both client and server can communicate concurrently
- As close to real-time as the TCP protocol can be

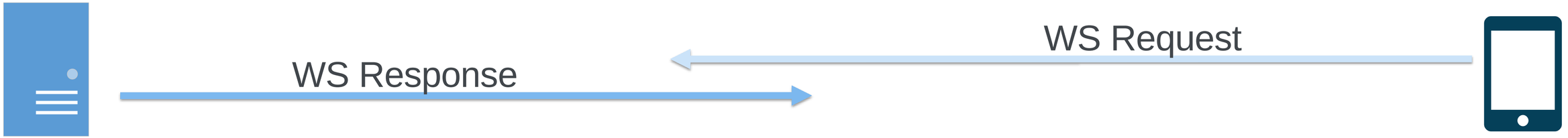
Web Sockets: “Now” Real-time




```
connect: function(url) {
    var ws = new WebSocket(“ws://url”);

    ws.onopen = function(e) {
        // ws open!
    };

    ws.onclose = function(e) {
        // ws closed!
    };

    ws.onmessage = function(e) {
        updateGui(e.message);
    }

    ws.onerror = function(e) {
        // ws error!
    };
};
```

Web Sockets (RFC6455): Line Protocol

1. Connection Handshake



HTTP Server Response (approx 120 bytes)

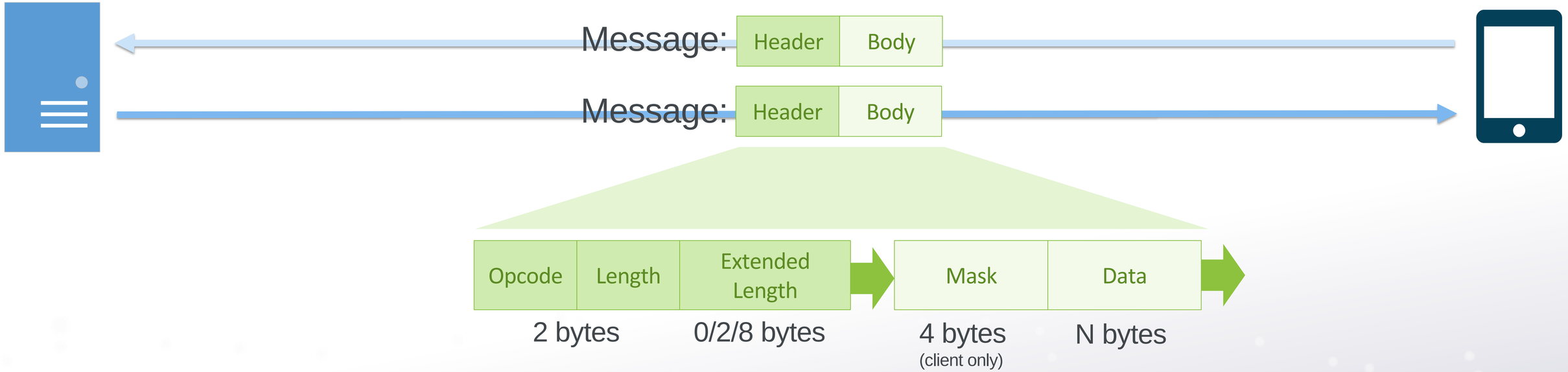
```
HTTP/1.1 101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: Ni3JjfWz9DjUSdDSVnQBJrpk24M=
```

HTTP Client Request (approx 560 bytes)

```
GET ws://www.yourdomain.com:80/ HTTP/1.1
Host: yourdomain.com:7200
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: http://www.yourdomain.com
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116
Safari/537.36
Accept-Encoding: gzip, deflate, sdch
...(abbreviated)…
```

Web Sockets (RFC6455): Line Protocol

2. Messaging



3. Closing Handshake

- Sends close opcode

WebSockets (RFC6455): API

- Constructor:
  - WebSocket(url, optionalProtocol)
- Methods/Properties
  - send(s)
  - close()
  - readyState (CONNECTING, OPEN, CLOSING, CLOSED)
- Event Handlers
  - onopen(e), onclose(e)
  - onmessage(e)
  - onerror(e)

```
var ws = new WebSocket("ws://websocket.org");
// or: var ws = new WebSocket("ws://websocket.org:8888", "ianaProtocolName");

ws.onopen = function (e)  {
      console.log("ws open…");
};

ws.onclose = function (e)  {
      console.log("ws closed…");
};

ws.onmessage = function (e)  {
      console.log("ws message received…");
      if (e.message == "pong") {
            ws.close();
      }
};

ws.onerror = function (e)  {
      console.log("ws error", e);
};

if (ws.readyState == "OPEN")  {
      ws.send("ping");
}
```

Web Sockets: Browser Version Support

- PC Browsers
  - Chrome – 29 and above
  - Firefox – 47 and above
  - IE –10 and above
  - Safari – 9.1 and above
- Device Browsers
  - Android Browser – 4.4 and above
  - Chrome for Android – 51 and above
  - IOS Safari – 9.2 and above (iPhone 4/IOS4 has a buggy implementation)
  - IE for Mobile – Windows Phone 8 and above
- Check:
  - https://websocketstest.com/
  - http://caniuse.com/#feat=websockets

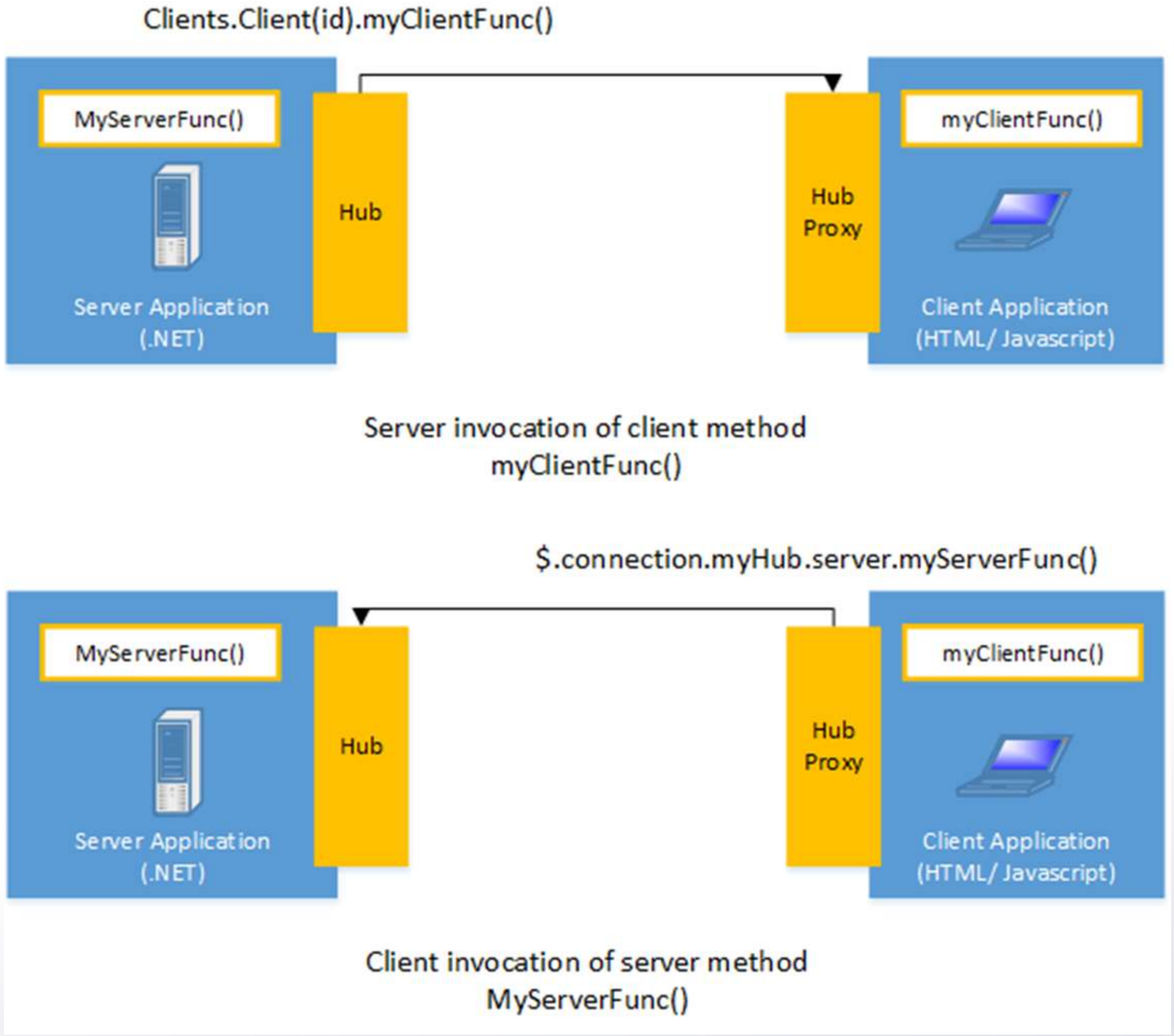
Demo

Part 3:
Real-time Web Frameworks and Sencha

Real-time Web Framework Recipes

1. CIY (Code It Yourself):
   - Client
     - JavaScript Web Sockets
     - Fallback to Long Polling when required
   - Server
     - Node.js / node-js-websocket
     - (.NET) SuperSocket and SuperWebSocket
2. Complete frameworks with Web Socket fallback options:
   - SignalR
   - socket.io/node.js

SignalR



http://www.asp.net/signalr/overview/getting-started/introduction-to-signalr

SignalR Features

- Transports with Fallbacks:
  1. WebSocket
  2. Server Sent Events
  3. AJAX Long Polling
- Rules
  - WebSockets are only be used when:
    - Client and server support them
    - For cross-domain connections, client needs to support CORS
  - JSONP uses Long Polling

SignalR

- Server framework deployable as:
  - ASP.NET Web Application
  - Standalone executable (eg Windows Service)
  - Azure “Web Role” or “Worker Role”
    - Scaleout using Azure Service Bus
    - https://www.asp.net/signalr/overview/performance/scaleout-with-windows-azure-service-bus
- Client framework should be included as a Sencha Resource:
  - SignalR script dependencies
  - Hubs file

SignalR C# Hub and Sencha Controller

Server C# Classes

```
public class SensorHub : Hub
{
    public Sensor getSensor(string id) {
        return new Sensor { id = id, name = "Sensor " + id, value = 0.0 };
    }

    public void notify(Sensor sensor)  {
        // Notify all clients…
        Clients.All.notifySensorUpdate(sensor);
    }
```

Client Sencha Controller

```
Ext.define('SignalrClient.controller.SensorController', {
    extend: 'Ext.app.Controller',
// config etc…
connectSignalR: function(button, e, eOpts) {
        var sensorHub = $.connection.sensorHub;
        sensorHub.url = "http://yourdomain.com:8088/signalr";

        sensorHub.client.notifySensorUpdate = function (sensor) {
            // Code to update Sensor gauges...
        };

        $.connection.hub.start().done(function () {
            // Code to show connection status...
        });

        var sensor1 = sensorHub.server.getSensor('1');
    },
```

SignalR C# Hub and Sencha Controller

Server C# Classes

```
    public void updateSensor(Sensor sensor) {
        // Update local Sensor in persistent store…
        notify (sensor)
    }
}

public class Sensor
{
    public string id { get; set; }
    public string name { get; set; }
    public double value { get; set; }
}
```

Client Sencha Controller

```
updateSensor: function(sensorHub, sensor) {
    sensorHub.server.updateSensor(sensor);
}

loadSensors: function(sensorHub) {
        var sensor1 = sensorHub.server.getSensor('1');
        var sensor2 = sensorHub.server.getSensor('2');
        var sensor3 = sensorHub.server.getSensor('3');

        // Code to add sensors to Store...
    }
});
```

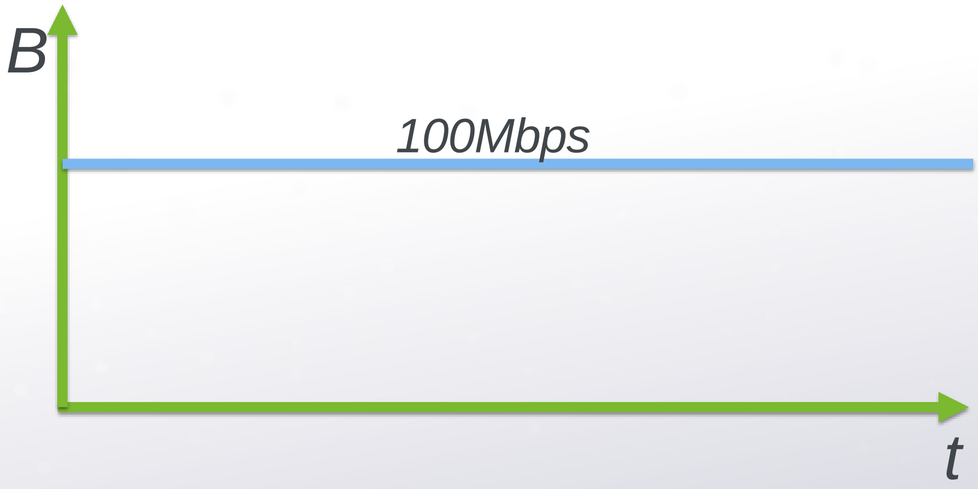
Part 4:
Real-time Web over Wireless Networks

Wireless Network Characteristics

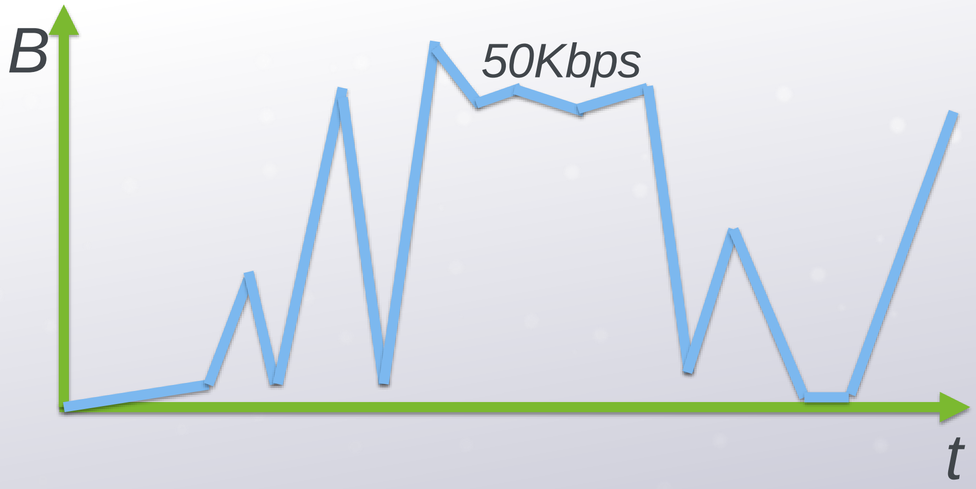
- Bandwidth is typically consistent if you are stationary
- But, bandwidth availability changes rapidly depending upon:
  - Obstructions between you and the nearest transmitter/receiver
  - Distance from the transmitter/receiver and number of users

Bandwidth Profiles

LAN



Wireless



Web Sockets to the Rescue?

- Full-duplex communication
- Minimal header size overhead
- “Now” Real-time

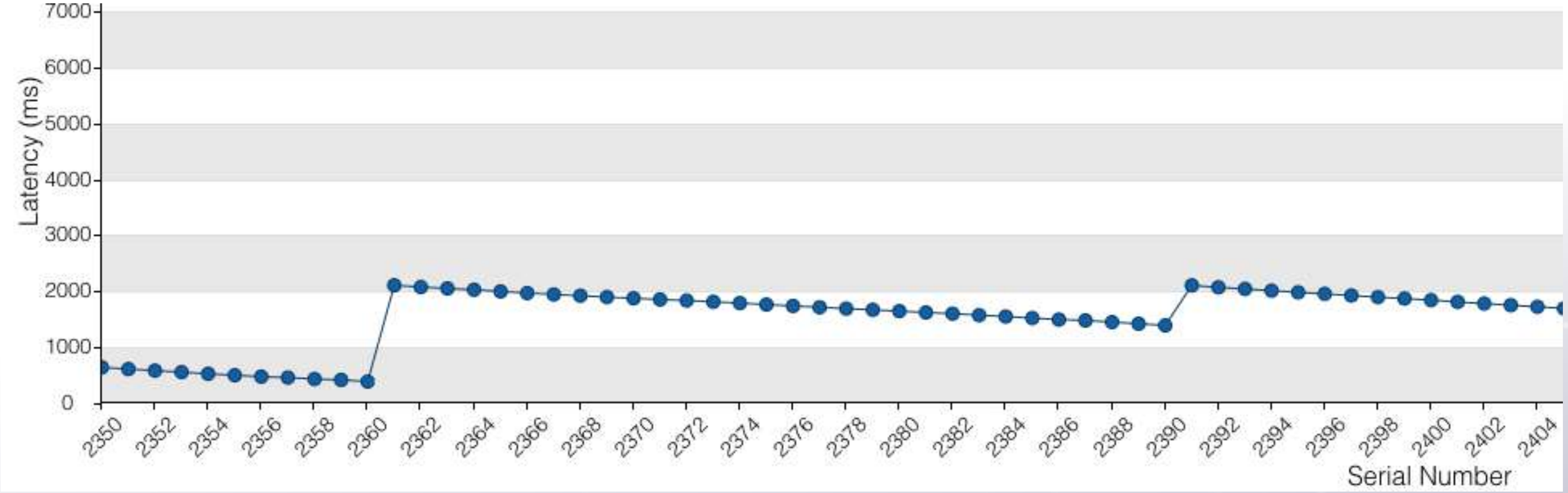
But…

- Web Server must allocate resources and maintain state for each WebSocket
- TCP is Stream Oriented – *everything* will be delivered *in order*

TCP and Real-time Communication

Wireless Simulation

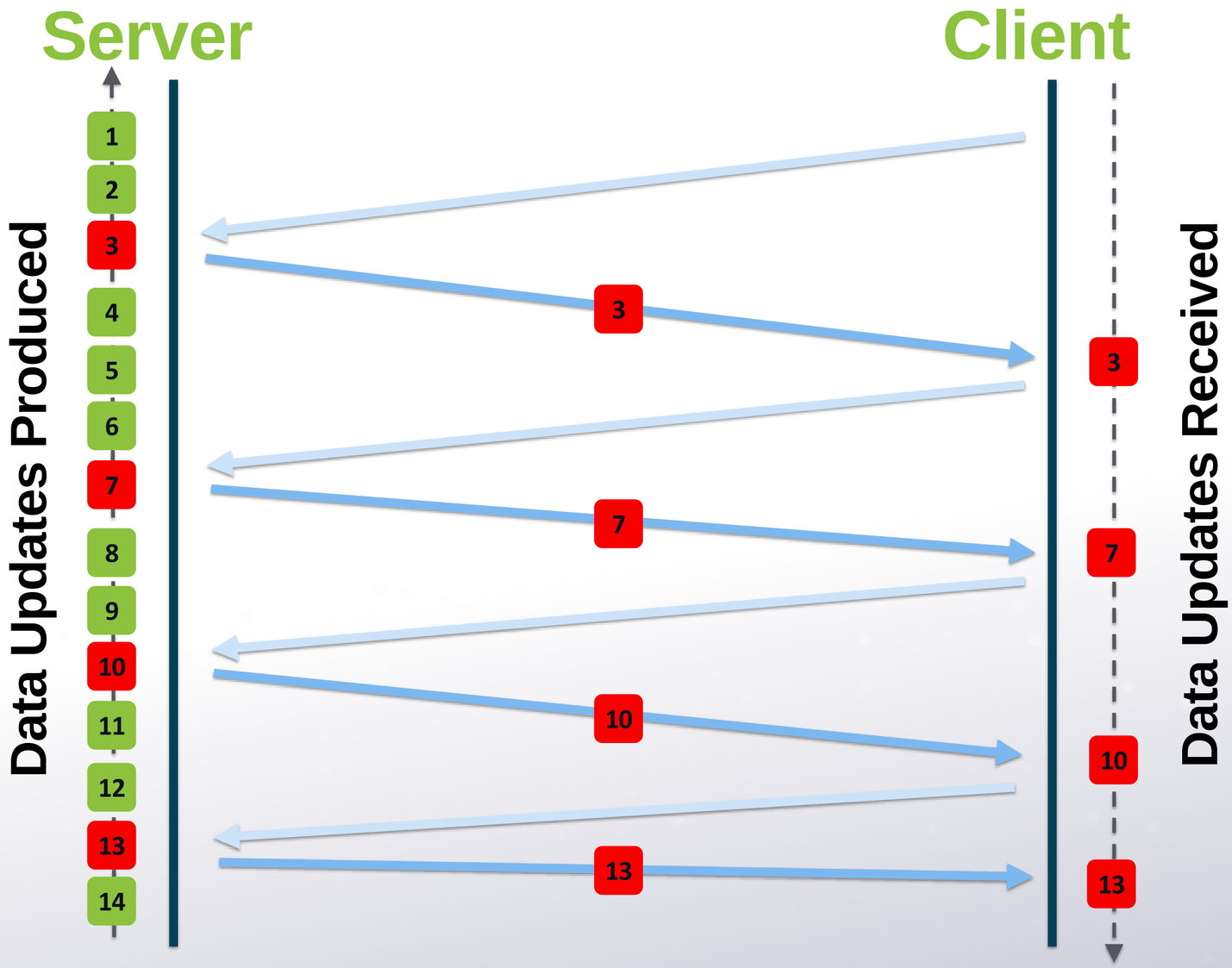
- 10 data updates per second
- 30% packet loss introduced (using “Clumsy”) on Ethernet connection



Delivering the TCP Stream

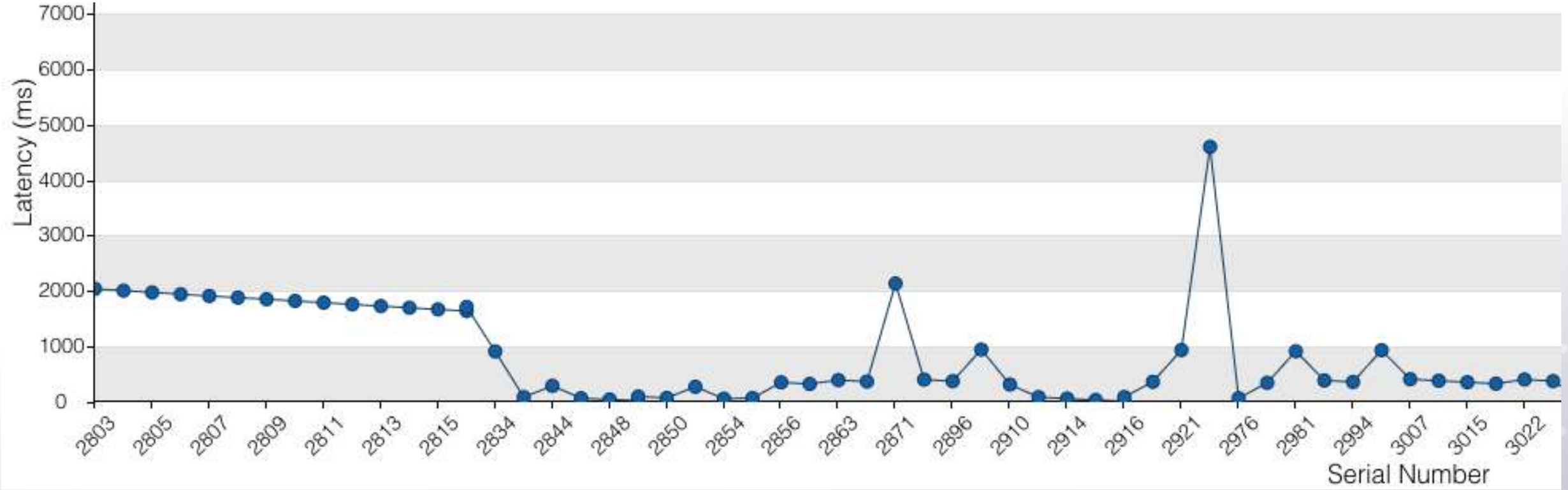
- Non time-critical data
  - Wait!
- Time-critical data
  - Throttle data update frequency according to bandwidth availability
  - How?
    - Consider a request/response mechanism similar to Long Polling, but over Web Sockets
    - In other words: “Server, please send me the latest update when you receive this request”

Request/Response Algorithm



Wireless Simulation

- Using a simple request/response algorithm over Web Sockets
- Average latency of each data update is reduced



Application Design Tips

- Prioritize the “real-time-ness” of data
  - Give priority to what the user is currently looking at
  - Give lower priority to keeping background stores etc up to date
- Register/de-register from data updates as the user navigates an app
- But remember users don’t want to switch to a new screen with old data on it

Application Testing Tips

- Measure your app’s bandwidth usage
- Test your application with:
  - Simulated packet loss
  - Socket disconnects
- SignalR offers a “connectionSlow” event
- Web Sockets can be queried for their “bufferedAmount”
- High frequency updates consume CPU on devices
- Watch the battery usage on small devices.

Summary

In Summary

- Use of Real-time web makes a highly responsive Sencha app
- Real-time web frameworks are easy to integrate into Sencha
- Web Sockets are great! But remember:
  - WebSockets require state and resources to be maintained on Server
  - TCP is a stream oriented protocol – *everything* will be delivered *in order*
- Test your application under poor network conditions
  - Packet loss, disconnects, jitter
- Design your data update algorithm to suit your applications requirements

SenchaCon 2016

Demo download:

www.schreuder.com.au/senchacon2016

Backup Slides

SenchaCon 2016

SignalR Hubs JavaScript Code

Available from: http://yourdomain:8088/signalr/hubs

```
proxies['sensorHub'] = this.createHubProxy('sensorHub');
        proxies['sensorHub'].client = { };
        proxies['sensorHub'].server = {
            getSensor: function (id) {
                return proxies['sensorHub'].invoke.apply(proxies['sensorHub'], $.merge(["getSensor"], $.makeArray(arguments)));
             },

            notify: function (sensor) {
                return proxies['sensorHub'].invoke.apply(proxies['sensorHub'], $.merge(["notify"], $.makeArray(arguments)));
             }
};
```

Web Sockets: Debugging in Chrome