# REAL-TIME WEB DEVELOPMENT

PART 1

Jim Paterson
James.Paterson@gcu.ac.uk



## HTTP Communication

- The web is based on the HTTP protocol
- Protocol defines the rules of conversation between client and server
- Strictly a **request-response** communication
- Client sends an HTTP request, and gets an HTTP response in return
- Request/response headers define details of communication
- Request/response bodies carry content of communication
- Response body typically contains a resource that the client requested

# Limitations of request-response

- Supports a single **communication model**
- Suitable for many use cases on the web but doesn't allow the web to offer a range of communication models that are common in other network scenarios and are required for some use cases
- Every piece of communication has to be initiated by the client
- Server can't "push" new data to the client when it becomes ready, client has to ask
- Clients can only communicate with the server, can't communicate directly with each other





# Communication models

- Server-push
  - Server can initiate communication to send data to all connected client(s) as soon as new data becomes available
  - Broadcast or directed to specific client
- Publish-subscribe
  - Senders of messages publish data to topics
  - Receivers subscribe to topics they want updates from
  - Senders don't know anything about receivers
  - Senders and receivers are clients
  - Updates sent to receivers as soon as available
- Peer-to-peer
  - Clients communicate directly without intermediary server

# Communication models and HTTP

| | Request-response | Server-push | Publish-subscribe | Peer-to-peer |
|---|---|---|---|---|
| Client-to-server | HTTP ✓ | HTTP ✗ | HTTP ✗ | |
| Server-to-client | HTTP ✗ | HTTP ✗ | HTTP ✗ | |
| Client-to-client | | | | HTTP ✗ |





# Real-time internet communication

- Real-time is about an experience that is happening “live” right now
- Open, bi-directional, synchronous, channel of communication
- Real-time is like a conversation where you immediately get the information and either of you can talk at any time
- Not actually possible to deliver data instantaneously, consider anything < 100ms to be “real-time” (based on research http://theixdlibrary.com/pdf/Miller1968.pdf)
- Traditional HTTP request-response can’t offer real-time, other communication models potentially can

# Real-time web

- Some providers implement real-time capabilities with standalone applications or browser plug-ins
- These can use specific network protocols that are not supported by web browsers
- The **real-time web** we are looking at here refers to the implementation of real-time capabilities within the browser, without the use of plug-ins





# Real-time web use cases

- Gaming
  - Multiplayer games
  - Leader boards and scores updated in real time
- Events
  - Live updates, e.g. scores are delivered in real time
  - Audience participation, e.g. voting
- Chat & Social
  - Messages typed and sent in real-time
  - Social streams pushed in real-time

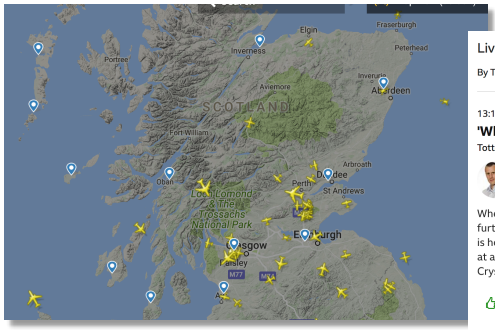
# Real-time web use cases

- Data & Content
  - Real time feeds of data like financial information, weather data, transport data etc.
  - Content pushed in real time, e.g. news, transport
- Notifications & Alerts
  - Particular events trigger notifications or alerts in real-time
- Collaboration
  - Real-time collaboration on documents, e.g. Google Docs
  - Real-time communications services like Skype

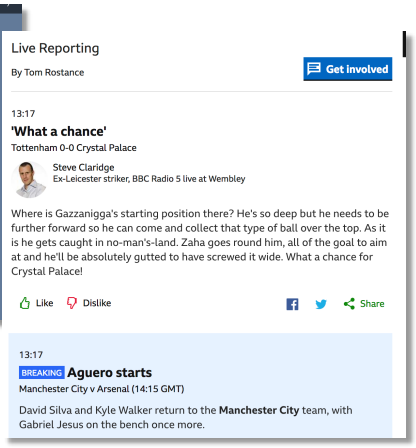
9



# Real-time web examples




FlightRadar24 – live flight locations
www.flightradar24.com




BBC sport live feed
www.bbc.co.uk

10

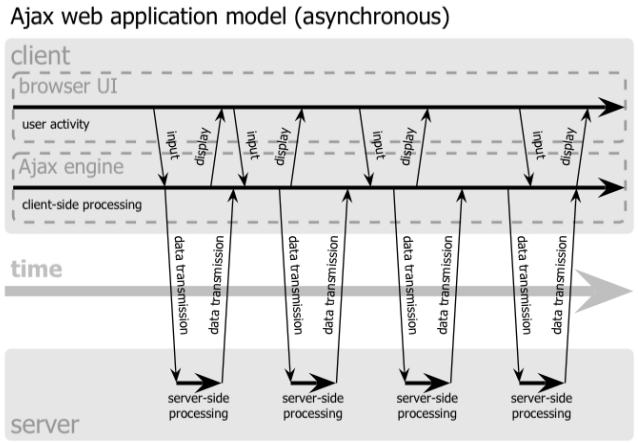
# Ajax – not real-time

- Ajax allows part of a web page to be updated without requiring a full page reload
- Client script invokes XMLHttpRequest (XHR) object
- Sends HTTP request
- Request-response communication, initiated by client
- So, not real-time – while page contents can be updated without reloading, the update only happens when the client specifically requests it





# Ajax application model



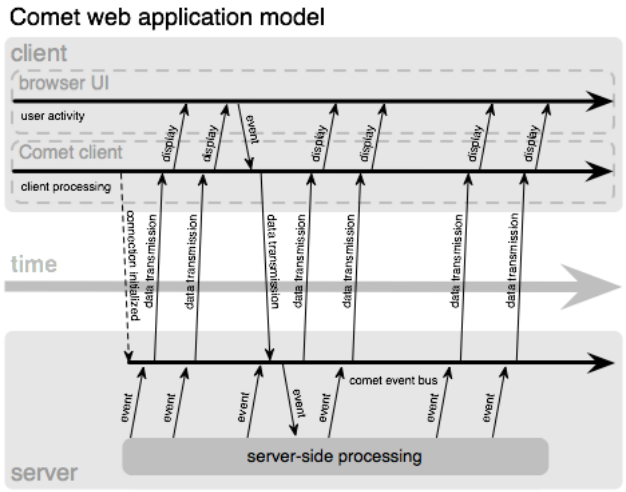
# Comet

- Alternative model for web communication proposed back in 2006 by Alex Russell
- Comet application – server pushes data to the client whenever new data is available
- Term not widely-used now, but can consider this the start of the real-time web
- Term describes "event-driven, server-push data streaming" programming style
- Not a specific technology - can be implemented in a number of different ways





# Comet application model



# Server push implementations

- A number of different mechanisms are available for implementing server push/Comet:
- Polling
- Long polling
- Forever frame
- Server-sent events
- WebSockets





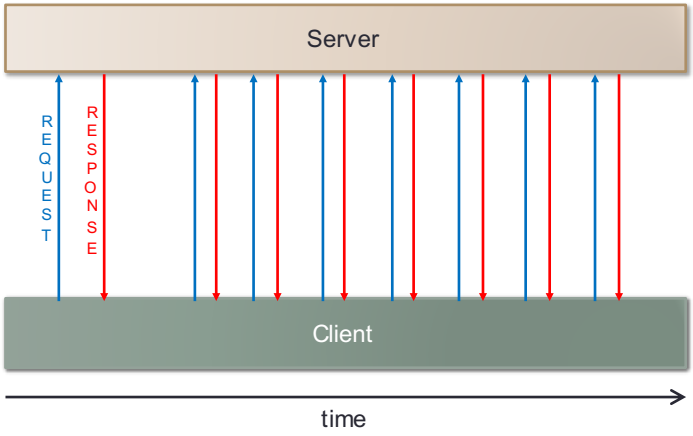
# Polling (periodic refresh)

- Browser can poll the server for updates every few seconds
- Use **setInterval** function in JavaScript to invoke XHR at specific intervals

```
setInterval(function(){
    self.sendData()
}, 5000);
```

- Requires only JavaScript and XHR support in browser, so works with most browsers, even older ones
- Doesn't require any support on the web server, which just sees standard HTTP requests and responds accordingly
- Still a widely-used solution

# Polling



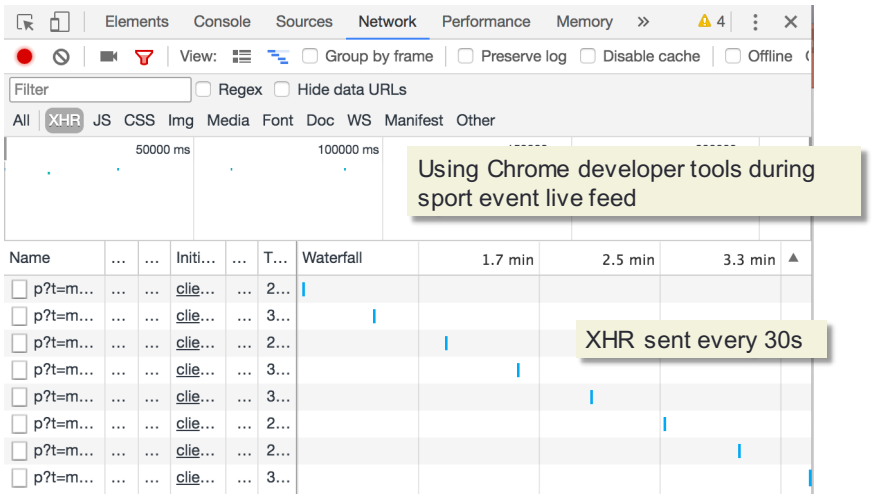





# Polling interval

- Polling is quite a crude solution that only emulates real-time communication
- Gives the user the impression of live updates
- However, updates are still initiated by the client
- If data changes infrequently many requests will be redundant as they will not retrieve any new data
- However, increasing **polling interval** increases latency perceived by user
- Need to find compromise for polling interval
- Scaling to large number of clients can cause excessive network traffic

# Polling – BBC Sport



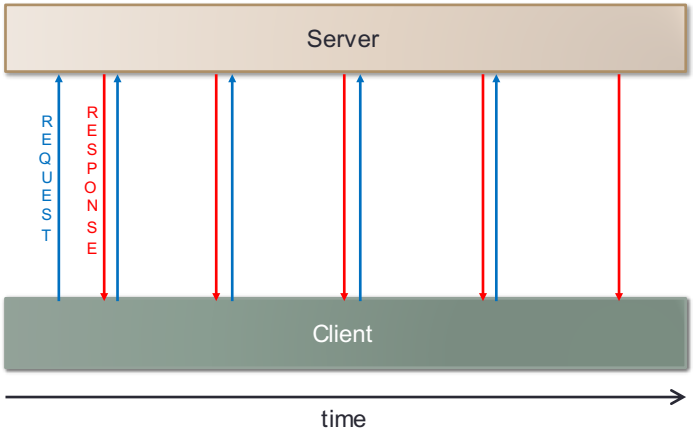





# Long polling

- Emulates server push a bit more accurately than polling
- The browser makes an async XHR request
- Server may wait for data before responding
- After processing of the response, the browser creates and sends another request
- Browser always keeps a request outstanding with the server
- Needs server-side support for the technique
- CometD was an early server implementation, other web servers also introduced support for long polling

# Long polling








# Forever frame

- The forever-frame technique uses HTTP 1.1 chunked encoding to establish a single, long-lived HTTP connection in a hidden iframe
- A hidden iframe element is opened in the browser after page load, establishing a long-lived connection inside the hidden iframe
- The server then continually sends script to the client which is immediately executed, providing a one-way realtime connection from server to client
- Requires browser and server support, mainly used with IE

# HTML 5 Server-Sent Events (SSE)

- Based on two main components:
  - The *text/event-stream* MIME type
  - The HTML5 EventSource interface with event listeners to receive messages
- Server application returns text/event-stream response and keeps HTTP connection open by including Connection: keep-alive header – easy to implement in server-side code
- SSE message data is text that contains a field name, a colon, and the field's value
- Comments—lines that starts with a colon followed by optional text—are ignored, but are useful as "keep alive" heartbeats to keep the HTTP connection open





# HTML5 SSE messages

- A single SSE message is made up of all the fields and data, parsed until an empty line (i.e. just a line feed)is received

```
event: message\n
data: Fernando Alonso\n
data: joined
\n
```

message with two lines of data

```
:First name
event: message\n
data: Fernando\n
\n
:Last name
data: Alonso\n
\n
```

stream consisting of two messages

Field names:
**event**: the event type, such as *message*, or another defined by your application
**data**: the field data itself
**retry**: an integer value indicating the reconnection time in case of a disconnect
**id**: message id value

# HTML5 SSE EventSource

- Consume data on client script using EventSource interface (browser must support this)
- This example consumes messages like the first one on previous slide – e.g. people joining a chat room

```
var people = new EventSource("http://www.example.com/chat");
people.onmessage = function (event) {
   var data = event.data.split('\n');
   var name = data[0];
   var action = data[1];
   console.log( name + " has " + action + " the room");
};
```

- What message should the server send when someone leaves the chat room?

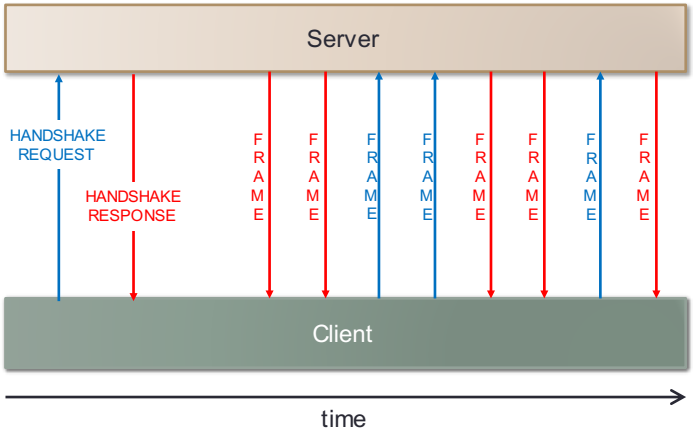




# WebSockets

- The WebSocket specification defines a full-duplex single socket connection over which messages can be sent between client and server
- Uses HTTP request/response as **handshake** to establish connection
- Then uses WebSocket protocol for true bidirectional communication
- Requires that the client and server both support the protocol (needs HTML 5 support in client)

# WebSockets








# WebSockets example

- Common example for WebSockets is a simple chatroom
- Look at example at this URL (full code available there) https://medium.com/@martin.sikora/node-js-websocket-simple-chat-tutorial-2def3a841b61
- WebSocket communication needs both a server and client component
- Client component is JavaScript within a web page
- WebSocket server may or may not be at the same site as the web server that served the web page – in this example they are separate
- WebSocket server here is written in Node.js, but focus here on the client code

# Client code outline

```
$(function () {
  // if user is running mozilla then use it's built-in WebSocket
  window.WebSocket = window.WebSocket || window.MozWebSocket;

  var connection = new WebSocket('ws://127.0.0.1:1337');

  connection.onopen = function () {
    // connection is opened and ready to use
  };

  connection.onerror = function (error) {
    // an error occurred when sending/receiving data
  };

  connection.onmessage = function (message) {
    // try to decode json (I assume that each message
    // from server is json)
    try {
      var json = JSON.parse(message.data);
    } catch (e) {
      console.log('This doesn\'t look like a valid JSON: ',
          message.data);
      return;
    }
    // handle incoming message
  };
});
```

- This code creates and opens a connection
- Also handles messages, assumed to be in JSON format
- JSON is parsed to variable *json*





# WebSocket handshake

- Running client code to open connection initiates a WebSocket handshake
- Client sends a regular HTTP request to the server
- **Upgrade** header in this request informs the server that the client wishes to establish a WebSocket connection
- If the server supports the WebSocket protocol, it agrees to the upgrade and sends an Upgrade header in the response
- Now that the handshake is complete the HTTP connection is replaced by a WebSocket connection that uses the same underlying TCP/IP connection
- At this point either party can starting sending data to the other as **frames**

# WebSocket handshake - headers

▼ **Request Headers** view parsed

```
GET ws://127.0.0.1:1337/ HTTP/1.1
Host: 127.0.0.1:1337
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: http://127.0.0.1:8887
Sec-WebSocket-Version: 13
```

▼ **Response Headers** view parsed

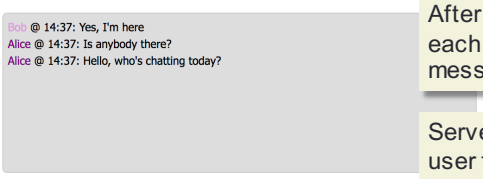
```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: +zfF6pqkdyMtOoRMm1BMH8rf9l0=
Origin: http://127.0.0.1:8887
```





# Chat room web page


After connection made – all users see a screen like this



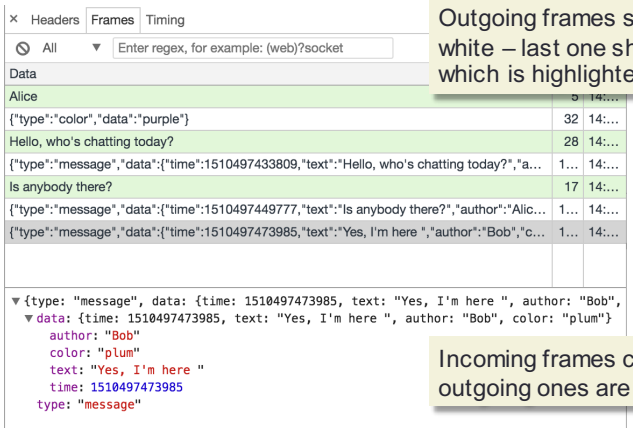

After Alice and Bob have connected, each from their own browser, and sent messages – this is Alice's screen

Server assigns a different colour to each user for message display

Message sent by one user is immediately seen by all connected users

# Chat room messages

- Look at frames in dev tools (Alice's browser)




Outgoing frames shown green, incoming white – last one shown is an incoming which is highlighted and inspected

Incoming frames carry JSON data, outgoing ones are simple text





# Client code message processing

```
if (json.type === 'color') {
  myColor = json.data;
  status.text(myName + ': ').css('color', myColor);
  input.removeAttr('disabled').focus();
  // from now user can start sending messages
} else if (json.type === 'history') { // entire message history
  // insert every single message to the chat window
  for (var i=0; i < json.data.length; i++) {
  addMessage(json.data[i].author, json.data[i].text,
      json.data[i].color, new Date(json.data[i].time));
  }
} else if (json.type === 'message') { // it's a single message
  // let the user write another message
  input.removeAttr('disabled');
  addMessage(json.data.author, json.data.text,
             json.data.color, new Date(json.data.time));
} else {
  console.log('Hmm..., I\'ve never seen JSON like this:', json);
}
```

- Message types can be *color*, *history*, *message*
- Content used to update CSS or content of DOM elements

## WebSocket chat example - summary

- Example shows features typical of a client-side application
- Uses *window.WebSocket* DOM object that is available if browser supports WebSockets
  - Connecting
  - Callbacks that are executed when message received
  - Sending messages using connection.send (see full source)
- Also has server component that listens for connections, responds to messages and constructs messages in JSON format (see full source code – it's in Node.js, which uses JavaScript, so should be fairly easy to understand)





## Raw WebSockets - limitations

- Example in previous slides is a raw WebSockets application
- Written assuming WebSockets supported
- Shows error message otherwise, but application will then simply not work
- All logic for distributing data to connected users is explicitly coded
- If we want to implement a specific communication model such as pub-sub then we have to code the logic for this

# Real-time frameworks

- A number of frameworks are available that support more useful real-time application development
  - **Automated selection of transport** – if WebSocket support not available (on client or server) will fall back to other transports such as SSE, long polling, to get application working
  - **Implementation of communication models**, such as pub-sub, on top of WebSockets
- Need to have server and client components
- Examples (there are many others):
  - **Socket.io** (https://socket.io) - Node/JavaScript, uses Redis for pub-sub
  - **SignalR** (https://www.asp.net/signalr) - ASP.NET/JavaScript





# Aside – HTTP 2 Server Push

- Newest version of HTTP protocol includes a capability called Server Push
- **Caution** – this is a different meaning of the term "server push", not related to real-time web applications
- HTTP 2 focuses on improving page load performance
- HTTP 2 allows server to push resources, e.g CSS files, to browser, without the browser explicitly requesting them, to speed page loading
- Doesn't push data to an application running in the browser

# Summary

- *Limitations of HTTP*
- *Communication models*
- *Real-time use cases and examples*
- *Ajax vs Server push (Comet)*
- *Server push transports – polling, long polling, forever frame, server-sent events, WebSocket*
- *WebSockets in action*