# **Middleware Architectures 2**

### **Lecture 6: Protocols for the Realtime Web**

### doc. Ing. Tomáš Vitvar, Ph.D.

tomas@vitvar.com • @TomasVitvar • https://vitvar.com



Czech Technical University in Prague
Faculty of Information Technologies • Software and Web Engineering • https://vitvar.com/lectures







Modified: Mon Feb 14 2022, 22:50:48 Humla v1.0

# **Overview of APIs and Protocols**

	XMLHttpRequest	Server-Sent Events	WebSocket
Request streaming	no	no	yes
Response streaming	limited	yes	yes
Framing mechanism	HTTP	event stream	binary framing
Binary data transfers	yes	no (base64)	yes
Compression	yes	yes	limited
App. transport protocol	HTTP	HTTP	WebSocket
Net. transport protocol	TCP	TCP	TCP

APIs and Protocols

### **Overview**

- Streaming and Long-polling
- WebSocket Protocol

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 3 -

# **Pushing and Polling**

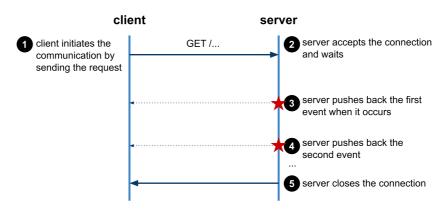
# Polling are there new data? no are there new data? no are there new data? yes

# open persistent connection Client Server new data

- Conceptual basis in messaging architectures
  - event-driven architectures (EDA)
- HTTP is a request-response protocol
  - response cannot be sent without request
  - server cannot initiate the communication
- **Polling** client periodically checks for updates on the server
- **Pushing** updates from the server (also called COMET)
  - = long polling server holds the request for some time
  - = streaming server sends updates without closing the socket

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

# **HTTP Streaming**



- server deffers the response until an event or timeout is available
- when an event is available, server sends it back to client as part of the response; this does not terminate the connection
- server is able to send pieces of response w/o terminating the conn.
  - using transfer-encoding header in HTTP 1.1
  - using End of File in HTTP 1.0

(server omits content-length in the response)

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 5 -

# **Chunked Response**

- Transfer encoding chunked
  - It allows to send multiple sets of data over a single connection
  - a chunk represents data for the event

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

25
This is the data in the first chunk

1C
9 and this is the second one
```

- Each chunk starts with hexadecimal value for length
- End of response is marked with the chunk length of 0
- Steps:
  - server sends HTTP headers and the first chunk (step 3)
  - server sends second and subsequent chunk of data (step 4)
  - server terminates the connection (step 5)

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

# **Issues with Chunked Response**

- Chunks vs. Events
  - chunks cannot be considered as app messages (events)
  - intermediaries might "re-chunk" the message stream
    - $\rightarrow$  e.g., combining different chunks into a longer one
- Client Buffering
  - clients may buffer all data chunks before they make the response available to the client application
- HTTP streaming in browsers
  - Server-sent events

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

-7-

# **XHR Polling**

• Client is making a period checks

```
function checkUpdates(url) {
   var xhr = new XHMLHttpRequest();
   xhr.open('GET', url);
   xhr.onload = function() { ... };
   xhr.send()
}
setInterval(function() { checkUpdates('/updates', 60000)});
```

- When new data is available, data is returend
- When no data is available, the response is empty
- Simple to implement but very inefficient
- Polling is expensive when interval is small
  - Each XHR request is a standalone HTTP request
  - HTTP incurs ~850 bytes of overhead for request/response headers
  - For example, 10 000 clients, each polling with 60 seconds interval:

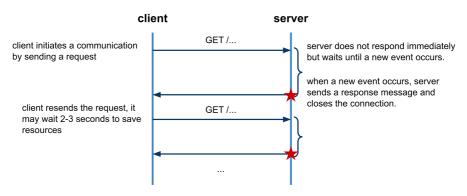
```
(850 \text{ bytes } \times 8 \text{ bits } \times 10000)/60 \text{ seconds} = 1.13 \text{ Mbps}
```

Server process 167 requests per second at a rate of 1.13 Mbps throughput

- Message latency
  - → Depends on the interval, maximum is 60 seconds
  - → Decreasing the interval will put more overhead on the server

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

# **XHR Long Polling**



### Server holds long-poll requests

- server responds when an event or a timeout occurs
- saves computing resources at the server as well as network resources
- can be applied over HTTP persistent and non-persistent communication

### Advantages and Issues

- Better message latency when interval is not constant
- concurrent requests processing at the server
- Too many messages may result in worse results that XHR polling

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 9 -

### **Server-Sent Events**

### • W3C specification

- part of HTML5 specs, see
- API to handle HTTP streaming in browsers by using DOM events
- transparent to underlying HTTP streaming mechanism
  - $\rightarrow$  can use both chunked messages and EOF
- same origin policy applies

### EventSource interface

- event handlers: onopen, onmessage, onerror
- constructor EventSource(url) creates and opens the stream
- method close() closes the connection
- attribute readyState
  - → CONNECTING The connection has not yet been established, or it was closed and the user agent is reconnecting.
  - $\rightarrow$  OPEN The user agent has an open connection and is dispatching events as it receives them.
  - $\rightarrow$  CLOSED The conn. is not open, the user agent is not reconnecting.

# **SSE Example**

Initiating EventSource

```
if (window.EventSource != null) {
   var source = new EventSource('your_event_stream.php');
} else {
   // Result to xhr polling :(
}
```

Defining event handlers

```
source.addEventListener('message', function(e) {
    // fires when new event occurs, e.data contains the event data
}, false);

source.addEventListener('open', function(e) {
    // Connection was opened
}, false);

source.addEventListener('error', function(e) {
    if (e.readyState == EventSource.CLOSED) {
        // Connection was closed
    }
}, false);
```

- when the conn. is closed, the browser reconnects every  $\sim$ 3 seconds  $\rightarrow$  can be changed using retry attribute in the message data

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 11 -

### **Event Stream Format**

- Format
  - response's content-type must be text/event-stream
  - every line starts with data:, event message terminates with 2 \n chars.
  - every message may have associated id (is optional)

```
id: 12345\n
data: first line\n
data: second line\n\n
```

JSON data in multiple lines of the message

- Changing the reconnection time
  - default is 3 seconds

```
1 | retry: 10000\n
2 | data: hello world\n\n
```

Lecture 6: Protocols for the Real time Web, CTU Summer Semester 2021/2022, @TomasVitvar Semester 2021/2022,

– 12 –

# **Auto-reconnect and Tracing**

- When a connection is dropped
  - EventSource will automatically reconnect
  - It may advertise the last seen message ID
    - → The client appends Last-Event-ID header in the reconnect request
  - The stream can be resumed and lost messages can be retransmitted
- Example

```
id: 43
    data: ...
    => Request (after connection is dropped)
    GET /stream HTTP/1.1
    Host: example.com
    Accept: text/event-stream
    Last-Event-ID: 43
10
    <= Response
    HTTP/1.1 200 OK
11
    Content-Type: text/event-stream
13
    Connection: keep-alive
    Transfer-Encoding: chunked
15
    id: 44
16
17
    data: ...
```

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 13 -

# **SSE Server-side implementation**

- Java Servlet
  - method doGet

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
3
4
        // set http headers
        resp.setContentType("text/event-stream");
resp.setHeader("cache-control", "no-cache");
        // current time in milliseconds
long ms = System.currentTimeMillis();
9
10
11
        // push data to the client for 20 seconds
12
        // client should reconnect when the connection is closed
        13
14
15
            resp.getWriter().flush();
16
            17
18
19
            } catch (InterruptedException e) {
20
                // do nothing;
21
            }
22
        }
    }
```

Lecture 6: Protocols for the Real time Web, CTU Summer Semester 2021/2022, @TomasVitvar Semester 2021/2022,

– 14 –

### **Streams API**

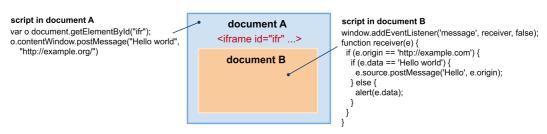
- An experimental technology, a living standard
- JavaScript API to programmatically access streams of data
  - Before, the whole resource had to be read
  - Now, you can process raw data as soon as it is available
    - $\rightarrow$  No need to generate string or buffer
    - → Detect streams start and end
    - → Chain streams together
    - → Handle errors, cancel streams
- Streams usage
  - Responses can be available as streams using Fetch API
    - → Response body returned by a fetch request and exposed as ReadableStream
    - → You can read it using ReadableStream.getReader()

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 15 -

# **Other Technologies**

Cross-document messaging



- The use of Cross Document Messaging for streaming
  - 1. The client loads a streaming resource in a hidden iframe
  - 2. The server pushes a JavaScript code to the iframe
  - 3. The browser executes the code as it arrives from the server
  - 4. The embedded iframe's code posts a message to the upper document

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

\_ 16 -

### **Overview**

- Streaming and Long-polling
- WebSocket Protocol

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

\_ 17 \_

### WebSocket

- Specifications
  - IETF defines
  - W3C defines
- Design principles
  - a new protocol
    - → browsers, web servers, and proxy servers need to support it
  - a layer on top of TCP
  - bi-directional communication between client and servers
    - → low-latency apps without HTTP overhead
  - Web origin-based security model for browsers
    - $\rightarrow$  same origin policy, cross-origin resource sharing
  - support multiple server-side endpoints
- Two phases
  - Handshake as an **upgrade** of a HTTP connection
  - data transfer the protocol-specific on-the-wire data transfer

Lecture 6: Protocols for the Real time Web, CTU Summer Semester 2021/2022, @Tomas Vitvar Semester 2021/2022,

– 18 –

# Handshake - Request

### Request

 client sends a following HTTP request to upgrade the connection to WebSocket

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 7
```

- − Connection − request to upgrade the protocol
- Upgrade protocol to upgrade to
- Sec-WebSocket-Key a client key for later validation
- Sec-WebSocket-Origin origin of the request
- Sec-WebSocket-Protocol list of sub-protocols that client supports (proprietary)

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 19 -

# Handshake - Response

### Response

- server accepts the request and responds as follows

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+x0o=
Sec-WebSocket-Protocol: chat
```

- → 101 Switching Protocols status code for a successful upgrade
- $\rightarrow$  Sec-WebSocket-Protocol a sub-protocol that the server selected from the list of protocols in the request
- → Sec-WebSocket-Accept a key to prove it has received a client WebSocket handshake request
- Formula to compute Sec-WebSocket-Accept

```
1 | Sec-WebSocket-Accept = Base64Encode(SHA-1(Sec-WebSocket-Key +
2 | "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"))
```

- $\rightarrow$  SHA-1 hashing function
- $\rightarrow$  Base64Encode Base64 encoding function
- $\rightarrow$  "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" magic number

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

### **Data Transfer**

### • After successful handshake

- socket between the client and the "resource" at the server is established
- client and the server can both read and write from/to the socket
- No HTTP headers overhead

### • Data Framing

- Data transmitted in TCP packets (see RFC6455: )
- Contains payload length, closing frame, ping, pong, type of data (text/binary), etc. and payload (message data)

0	1	2	3		
0 1 2 3 4 5 6 7 8	9 0 1 2 3 4 5	6 7 8 9 0 1 2 3 4	5 6 7 8 9 0 1		
F   R   R   R   opcode   M     I   S   S   S   (4)   A     N   V   V       S       1   2   3     K	Payload len (7)	Extended paylo (16/ (if payload le	64)		
Extended payload length continued, if payload len == 127					
!		Masking-key, if M	ASK set to 1		
Masking-key (continued)		Payload	Data		
: Payload Data continued :					
Payload Data continued					
T			<del></del>		

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 21 -

# Frame and Message

### Frame

- The smallest unit of communication, composed of a header and a paylod
- Payload carries all or part of a message

### Message

- A sequence of frames that map to a logical application message
- Messages decomposition into frames is done by client and server framing code
- Application is unaware of frames but only application messages

### • Frame on-the-wire

- FIN indication whether the frame is final in the message
- opcode type of frame: (text, binary, control frame, ping, pong)
- Mask bit indicates if the payload is masked (from client to server only)
- Payload length
- Masking key contains 32-bit key to mask the payload (if mask bit is set)
- Payload application data

# **Head-of-line Blocking**

- Head-of-line blocking recall
  - Second request needs to wait for the first request to finish
  - See HTTP Pipelining in AM1 for more details
- WebSocket
  - There is no head-of-line blocking problem
  - Messages can be split into more frames
  - But frames cannot be interleaved
- ⇒ A large message with more frames may block frames of other messages
- You need to be careful of payload size of each message
  - You should split large message into multiple application messages
- Each WebSocket connection requires a dedicated TCP connection
  - Problem with HTTP/1.x due to a restricted number of connections per origin

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 23 -

### WebSocket Infrastructure

- HTTP optimized for short transfers
  - intermediaries are configured to timeout idle HTTP connections quickly
  - This can be a problem for long-lived WebSocket connections
- We cannot control
  - Client's network, some networks may block WebSocket traffic
    - $\rightarrow$  We need a fallback strategy
  - Proxies on the external network
    - → TLS may help, tunneling over secure end-to-end connection, WebSocket traffic can bypass intermediaries
- We can control our infrastructure
  - For example, set tunnel timeout to 1 hour on HAProxy

```
1 defaults http
2 timeout connect 30s
3 timeout client 30s
4 timeout server 30s
5 timeout tunnel 1h
```

### WebSocket API

- Client-side API
  - clients to utilize WebSocket, supported by Chrome, Safari
  - Hides complexity of WebSocket protocol for the developer
- JavaScript example

```
// ws is a new URL schema for WebSocket protocol; 'chat' is a sub-protocol
var connection = new WebSocket('ws://server.example.org/chat', 'chat');

// When the connection is open, send some data to the server
connection.onopen = function () {
    // connection.protocol contains sub-protocol selected by the server
    console.log('subprotocol is: ' + connection.protocol);
    connection.send('data');
};

// Log errors
connection.onerror = function (error) {
    console.log('WebSocket Error ' + error);
};

// Log messages from the server
connection.onmessage = function (e) {
    console.log('Server: ' + e.data);
};

// closes the connection
connection.close()
```

Lecture 6: Protocols for the Realtime Web, CTU Summer Semester 2021/2022, @TomasVitvar

- 25 -