# 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



Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti



Modified: Mon May 02 2022, 06:37:10
Humla v1.0

# Overview of APIs and Protocols

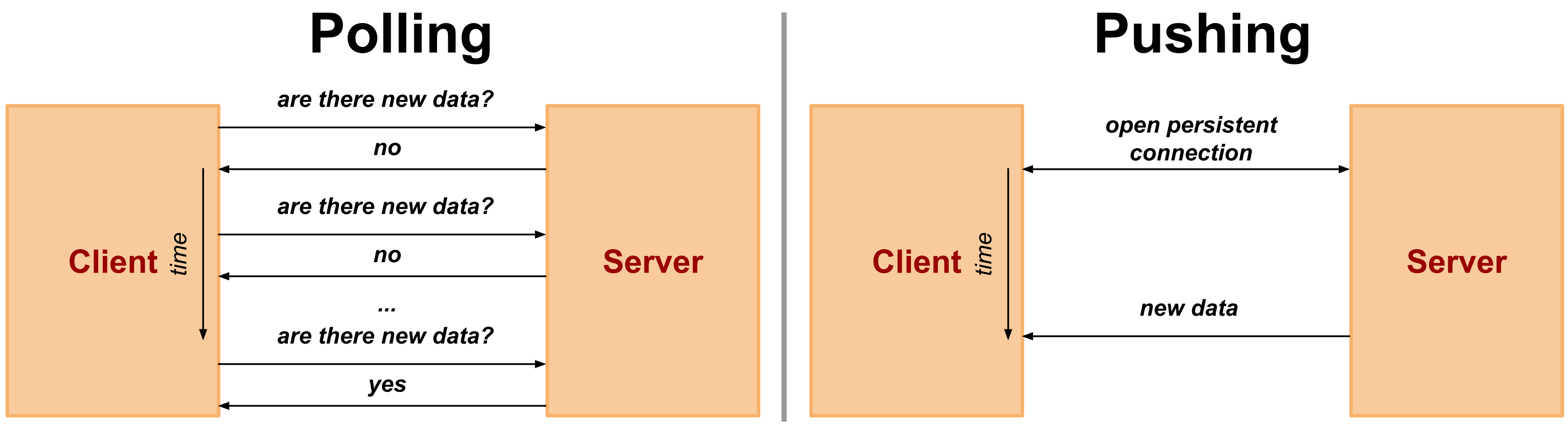
| | XMLHttpRequest | Server-Sent Events | WebSocket |
|---|---|---|---|
| Request streaming | no | no | yes |
| Response streaming | limited | yes | yes |
| Framing mechanism | HTTP | event stream | binary framing |
| Binary data transfers | yes | no (base64) | yes |
| Compression | yes | yes | limited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

APIs and Protocols

# Overview

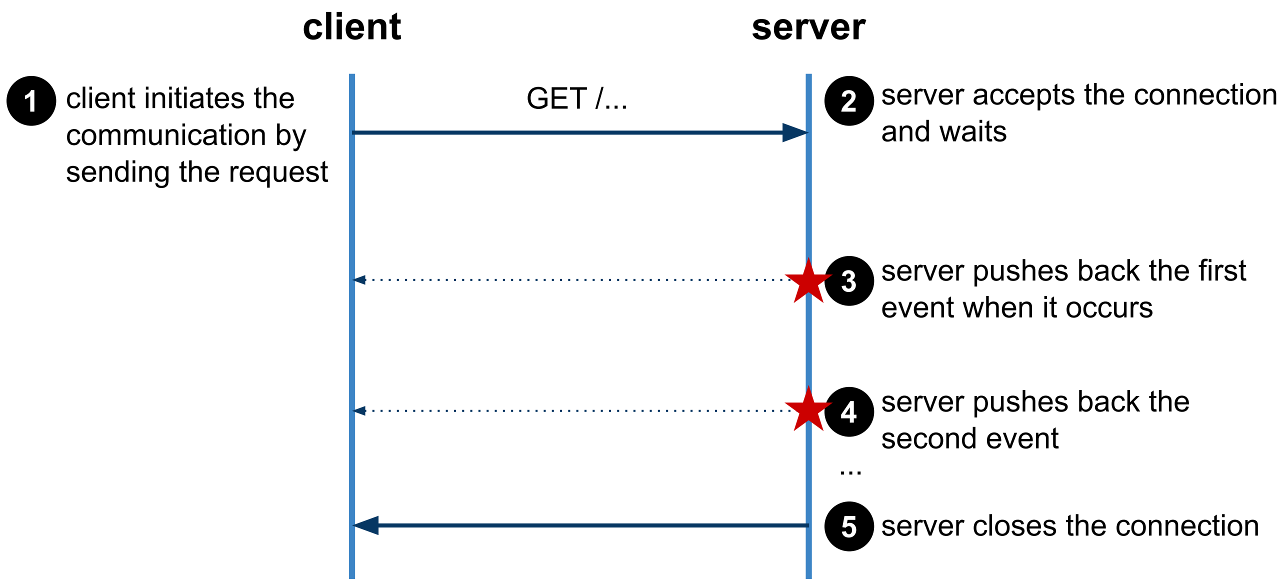
- Streaming and Long-polling
- WebSocket Protocol

# Pushing and Polling




- 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

# 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-lenght` *in the response)*

# 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
and this is the second one

0
```

  - *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)*

# Issues with Chunked Response

- Chunks vs. Events
  - *chunks cannot be considered as app messages (events)*
  - *intermediaries might "re-chunk" the message stream*
    *→ 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*

# 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 reqquest 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:*

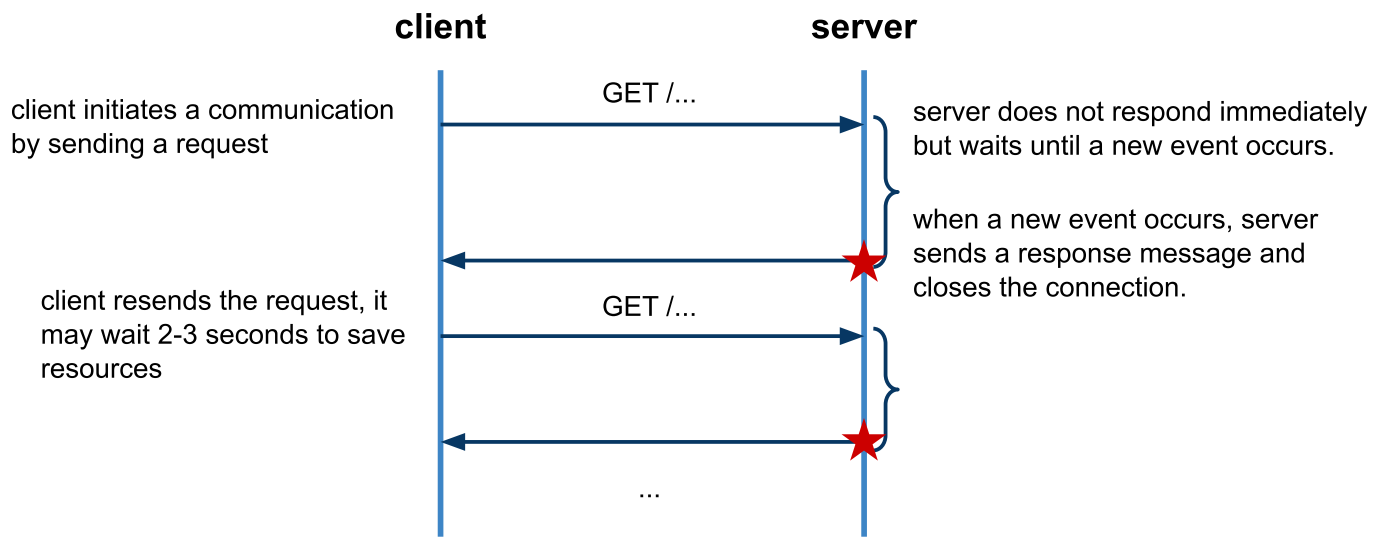
    `(850 bytes x 8 bits x 10000)/60 seconds = 1.13 Mbps`

    *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*

# 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*

# Server-Sent Events

- W3C specification
  - *part of HTML5 specs, see Server-Sent Events*
  - *API to handle HTTP streaming in browsers by using DOM events*
  - *transparent to underlying HTTP streaming mechanism*
    - → *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.*
    - → `OPEN` *– The user agent has an open connection and is dispatching events as it receives them.*
    - → `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 ~3 seconds*
    → *can be changed using* `retry` *attribute in the message data*

# 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

```
data: {\n
data: "msg": "hello world",\n
data: "id": 12345\n
data: }\n\n
```

- Changing the reconnection time
  - *default is 3 seconds*

```
retry: 10000\n
data: hello world\n\n
```

# 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

<= Response
HTTP/1.1 200 OK
Content-Type: text/event-stream
Connection: keep-alive
Transfer-Encoding: chunked

id: 44
data: ...
```

# SSE Server-side implementation

- Java Servlet
  – *method* doGet

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws IOException {

    // set http headers
    resp.setContentType("text/event-stream");
    resp.setHeader("cache-control", "no-cache");

    // current time in milliseconds
    long ms = System.currentTimeMillis();

    // push data to the client for 20 seconds
    // client should reconnect when the connection is closed
    while (System.currentTimeMillis() - ms < 20000) {
        resp.getWriter().print("data: servlet runs for " +
            (System.currentTimeMillis() - ms)/1000 + " seconds.\n\n");
        resp.getWriter().flush();
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            // do nothing;
        }
    }
}
```

# Streams API

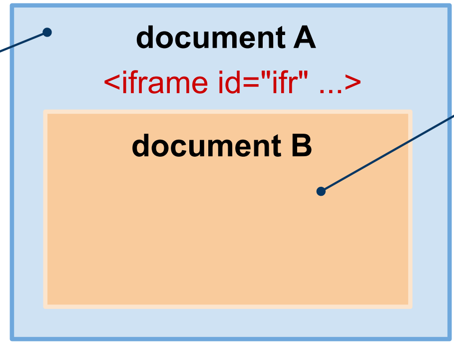
- An experimental technology, a living standard
- JavaScript API to programmatically access streams of data
  - *Before, the whole resource had to be read*
  - *Now, you can process raw data as soon as it is available*
    - → *No need to generate string or buffer*
    - → *Detect streams start and end*
    - → *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()`
    - → *You can cancel it using* `ReadableStream.cancel()`

# Other Technologies

- Cross-document messaging

**script in document A**

```
var o document.getElementById("ifr");
o.contentWindow.postMessage("Hello world",
    "http://example.org/")
```




**script in document B**

```
window.addEventListener('message', receiver, false);
function receiver(e) {
  if (e.origin == 'http://example.com') {
    if (e.data == 'Hello world') {
      e.source.postMessage('Hello', e.origin);
    } else {
      alert(e.data);
    }
  }
}
```

– *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*

# Overview

- Streaming and Long-polling
- WebSocket Protocol

# WebSocket

- Specifications
  - *IETF defines WebSocket Protocol*
  - *W3C defines WebSocket API*
- 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*
    - *→ same origin policy, cross-origin resource sharing*
  - *support multiple server-side endpoints*
- Custom URL scheme: `ws` and `wss` (TCP and TLS)
  - *WebSocket can be used over non-HTTP connections outside of browsers*
- Options to establish a WebSocket connection
  - *TLS and ALPN*
  - *HTTP/1.1 connection upgrade*

# Connection Upgrade – 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)*

# Connection Upgrade – Response

- Response
  - *server accepts the request and responds as follows*

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

→ `101 Switching Protocols` – *status code for a successful upgrade*

→ `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`

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

→ `SHA-1` – *hashing function*

→ `Base64Encode` – *Base64 encoding function*

→ `"258EAFA5-E914-47DA-95CA-C5AB0DC85B11"` – *magic number*

# Data Transfer

- After a successful connection upgrade
  – *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: Base Framing Protocol)*
  – *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| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

# 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
  - *Large messages can cause the head of line blocking on the client*
  - *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*
    - → *monitor amount of messages in the client's buffer*
    - → *send data when buffer is empty*
- Each WebSocket connection requires a dedicated TCP connection
  - *Problem with HTTP/1.x due to a restricted number of connections per origin*

# WebSocket Browser API

- Client-side API
  - *clients to utilize WebSocket, supported by all modern browsers*
  - *Hides complexity of WebSocket protocol for the developer*
- JavaScript example

```javascript
// 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()
```

# Avoid Head of Line Blocking

- Example code to monitor a client buffer size

```
var ws = new WebSocket('wss://example.com/socket');

ws.onopen = function () {
  subscribeToApplicationUpdates(function(evt) {
    if (ws.bufferedAmount == 0)
      ws.send(evt.data);
  });
};
```

  - *A call to* `send` *is asynchronous*
  - *Multiple calls to* `send` *will fill the client buffer*

# 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*
    - → *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*

```
defaults http
    timeout connect 30s
    timeout client  30s
    timeout server  30s
    timeout tunnel  1h
```