Web 2.0

Lecture 5: Protocols for the Realtime Web

doc. Ing. Tomáš Vitvar, Ph.D.
tomas@vitvar.com • @TomasVitvar • http://vitvar.com

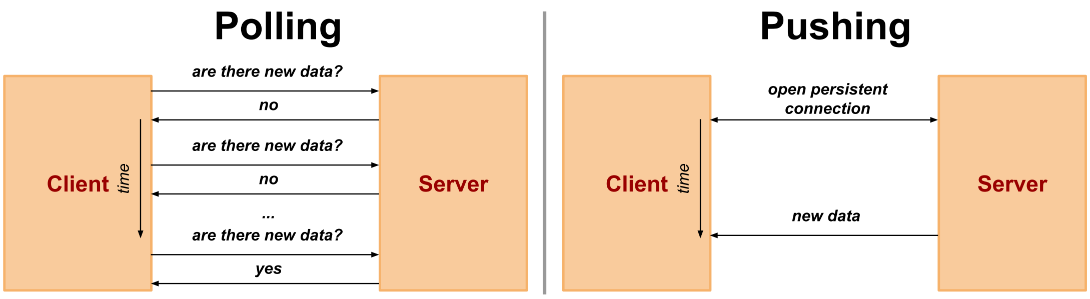
Czech Technical University in Prague
Faculty of Information Technologies • Software and Web Engineering • http://vitvar.com/courses/w20

Modified: Mon Apr 15 2019, 15:57:03
Humla v0.3

Overview

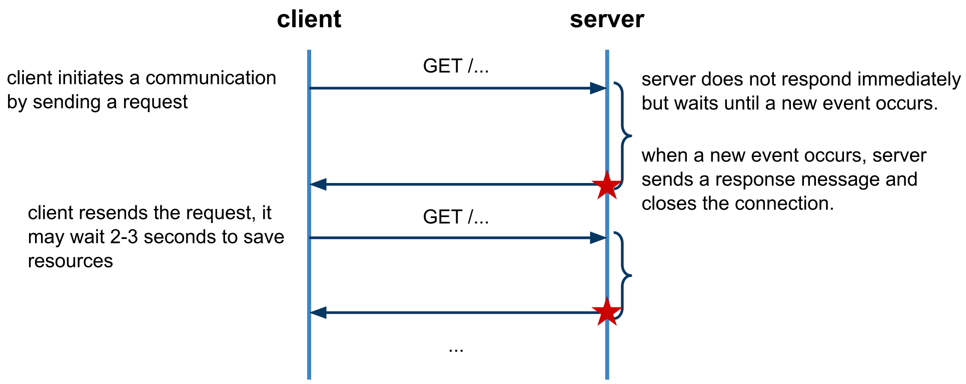
- Long-polling and Streaming
- WebSocket Protocol
- WebRTC

Pushing and Polling



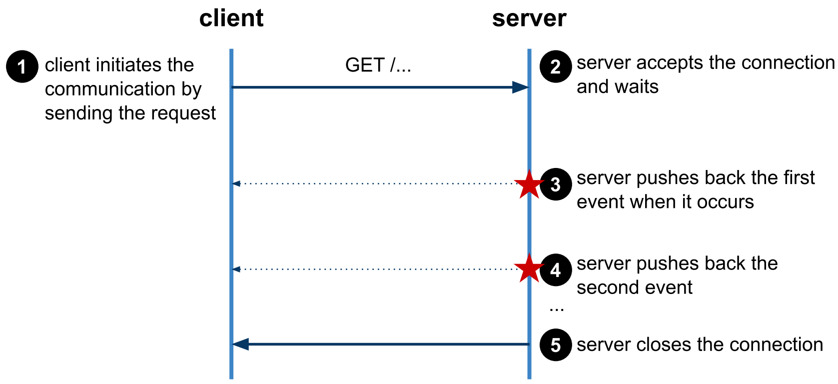
- Conceptual basis in messaging architectures
  – *event-driven architectures (EDA)*
- **HTTP is a request-response protocol**
  – *response cannot be sent without request*
  – *server cannot initiate the communication*
- **Polling** – client periodically checks for updates on the server
- **Pushing** – 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 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*
- Issues:
  – *maximum time of the request processing at the server*
  – *concurrent requests processing at the server*

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*

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.*

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
```

Server-side implementation

- Java Servlet
  - *method* `doGet`

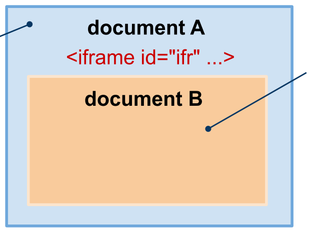
```java
public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws IOException {

    // 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;
        }
    }
}
```


Other Technologies

- Cross-document messaging

script in document A

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



script in document B

```javascript
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

- Long-polling and Streaming
- WebSocket Protocol
- WebRTC

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*
- Two phases
  – *Handshake – as an **upgrade** of a HTTP connection*
  – *data transfer – the protocol-specific on-the-wire data transfer*

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)*

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+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 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: 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 ...                |
+---------------------------------------------------------------+
```


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()
```

Sockets.IO

- Many options for streaming
  - *long-polling, streaming, iframe, WebSockets*
  - *Not all browsers support WebSockets*
  - *Socket.IO – a layer providing a unified API*
- Sockets.IO
  - *API and JavaScript implementation*
  - *checks the availability of WebSocket protocol*
    - → *fallback to long-polling or other technologies when not available*

```
// creates a new socket
var socket = new io.Socket();

// event handlers
socket.on('connect', function(){
  socket.send('hi!');
})
socket.on('message', function(data){
  alert(data);
})
socket.on('disconnect', function(){})
```


Streaming video

- Webcams, IP or USB
  - *Play video stream using RTSP or M-JPEG*
  - *RTSP (Realtime Streaming Protocol) defines sequences to control palying multimedia*
- Sample tasks
  - *Add video stream to a web page*
    - → `video` *HTML5 element*
  - *Capture frames from the camera and process them*
    - → *Capture frames in a specific format such as JPG*
    - → *Specific software to capture frames, typically OpenCV*
  - *Add annotation to video and expose as video stream*
    - → *Detect objects in pictures – machine learning/deep learning*
    - → *Mark objects and expose frames as video to the client*
    - → *Create RTSP stream by using e.g. GStreamer or FFMPEG*
    - → *Create stream of JPG images, so called M-JPEG and push them to the client*

M-JPEG

- M-JPEG - Motion JPEG
  - *Video compression format, each frame is represented as a JPEG image*
  - *Widely used by cameras today*
  - *Uses HTTP response stream of* `multipart/x-mixed-replace` *content type*
- Example HTTP response to a M-JPEG request

```
HTTP/1.1 200 OK
Content-Type: multipart/x-mixed-replace; boundary=imgboundary

--imgboundary
Content-Type: image/jpeg
Content-length: 5432

[image 1 encoded jpeg data]

--imgboundary
Content-Type: image/jpeg
Content-length: 54335

[image 2 encoded jpeg data]

...

```


Overview

- Long-polling and Streaming
- WebSocket Protocol
- WebRTC

WebRTC

- Web Real-Time Communication
  - *API to exchange media and arbitrary data between peers inside Web pages*
  - *It uses peer-to-peer principles*
  - *Supported by Google, Mozilla, Microsoft, Opera*
- Specifications
  - *WebRTC IETF Working Groups*
  - *WebRTC W3C Working Groups*
- History
  - *Google acquires company Global IP Solutions (GIPS) in 2010*
  - *GIPS developed underlying technology (codecs, echo cancellation techniques), released as open source*
  - *Google promoted the work around GIPS to W3C and IETF*

WebRTC Main Tasks

- Acquiring audio and video
  - *JavaScript API:* `MediaStream` *(aka getUserMedia)*
- Communicating audio and video
  - *JavaScript API:* `RTCPeerConnection`
- Communicating arbitrary data
  - *JavaScript API:* `RTCDataChannel`

GetUserMedia

- JavaScript code

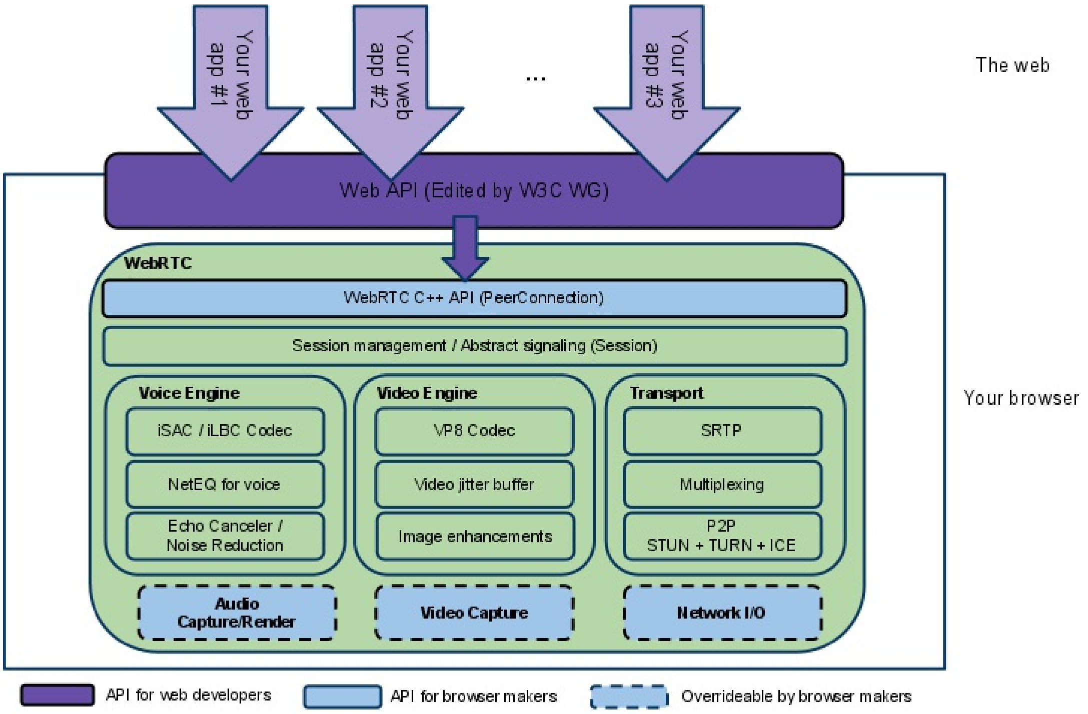
```
var constraints = {video: true};

function successCallback(stream) {
  var video = document.querySelector("video");
  video.src = window.URL.createObjectURL(stream);
}

function errorCallback(error) {
  console.log("navigator.getUserMedia error: ", error);
}

navigator.getUserMedia(constraints, successCallback, errorCallback);
```

- Constraints
  – *Control the contents of the MediaStream*
  – *Media type, resolution, frame rate*
- JavaScript app can read and manipulate the stream.
- It is also possible to acquire **audio** as well as **screen capture.**

RTCPeerConnection

- Allows to communicate media stream acquired by `getUserMedia`
  – *Video chat, audio chat, screen sharing*
- Some capabilities of `RTCPeerConnection`
  – *Signal processing*
  – *Code handling*
  – *Peer to peer communication*
  – *Security*
  – *Bandwidth management*

WebRTC Architecture



Communication

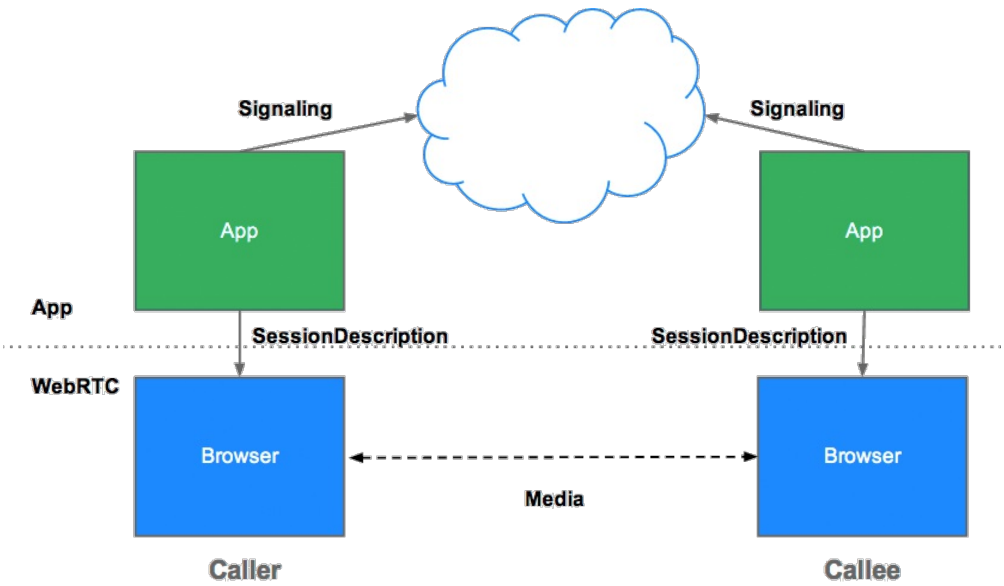
- Two phases
  1. ***Signaling***
     – *WebRTC defines* ***abstract signalling***
     – *apps can use any singaling protocol, can use any such as SIP, XMPP, or custom using XHR or Websockets*
  2. ***Exchange of real-time*** *data in peer-to-peer manner*
- Abstract signaling
  – *Need to exhange* ***session description*** *objects*
    → *Formats, codecs the peers want to use*
    → *Network information for peer-to-peer communication*
    → *This information is captured as* `RTCSessionDescription` *(also SDP) structure*
  – *Any messaging mechanism and protocol*

SIP and SDP

- Standards
  - *SIP – Session Initiation Protocol, protocol to establish and modify sessions.*
  - *SDP – Session Description Protocol, describes media for a session, defined in RFC4566 - Session Description Protocol*
- SDP Example

```
v=0
o=- 7614219274584779017 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE audio video
a=msid-semantic: WMS
m=audio 1 RTP/SAVPF 111 103 104 0 8 107 106 105 13 126
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:W2TGCZw2NZHuwlnf
a=ice-pwd:xdQEccP40E+P0L5qTyzDgfmW
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=mid:audio
a=rtcp-mux
a=crypto:1 AES_CM_128_HMAC_SHA1_80 inline:9c1AHz27dZ9xPI91YNfSlI67/EMkjHHIHORiClQe
a=rtpmap:111 opus/48000/2
...
```


Signaling Diagram



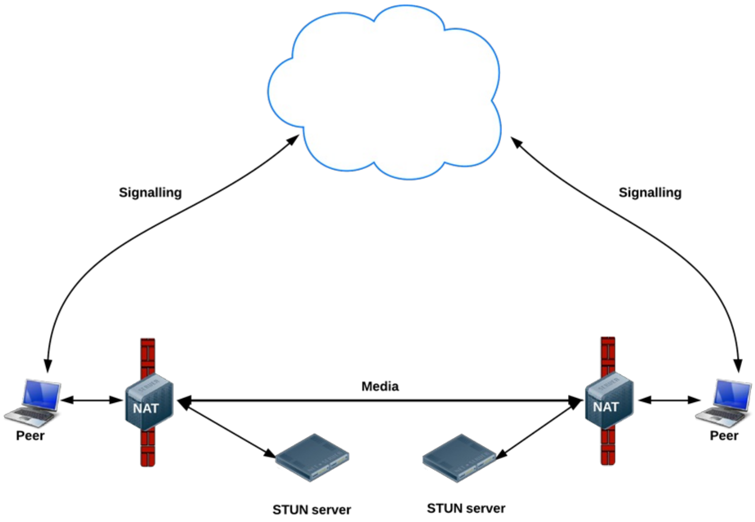
JavaScript Session Establishment (JSEP)

- JSEP is a protocol to create a session between two parties
  - *The interface needed by an application to deal with the negotiated local and remote session descriptions*
- JSEP steps between Alice and Bob
  1. *Alice creates an offer that contains her local SDP.*
  2. *Alice attaches that offer to* `RTCPeerConnection` *object.*
  3. *Alice sends the offer to a singaling server using custom-built mechanism (WebSocket, XHR, etc.)*
  4. *Bob receives Alice's offer from the signaling server*
  5. *Bob creates an answer using his local SDP.*
  6. *Bob attaches his answer along with Alice's offer to his own* `RTCPeerConnection` *object.*
  7. *Bob returns his answer to the singaling server.*
  8. *Alice receives Bob's offer from the singaling server.*

Interactive Connectivity Establishment

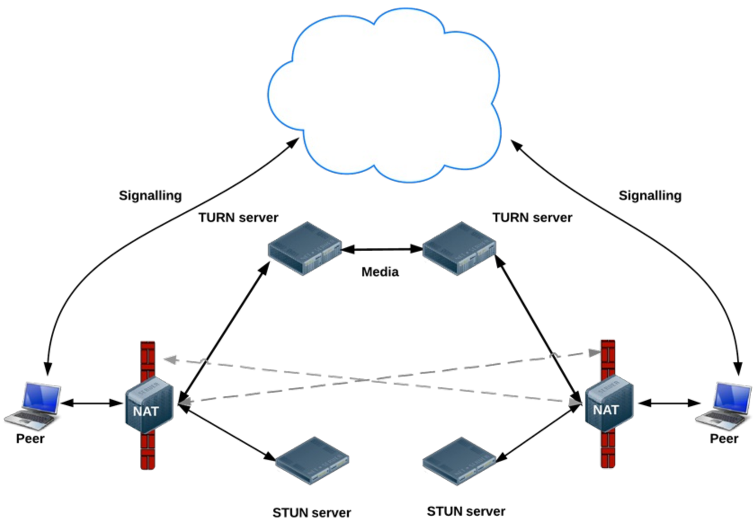
- ICE – Interactive Connectivity Establishment
  - *Allows WebRTC to overcome complexities of real-world networking*
  - *Finds the best path to connect peers such as*
    - *→ direct P2P communication.*
    - *→ by using STUN or TURN servers.*
- STUN – Session Traversal Utilities for NAT
  - *Allows to discover the presence of a NAT server.*
  - *Allows to discover the public IP address and a port that the NAT has allocated for UDP flows.*
  - *It is provided as a third-party network server (STUN server) located on the public side of the NAT.*
- TURN – Traversal Using Relays around NAT
  - *Communication relay for hosts behind NAT when STUN does not work.*

STUN



→ STUN is a simple server, cheap to run
→ Data flows peer-to-peer

TURN



→ a cloud fallback when peer-to-peer does not work
→ data sent via a relay server, uses server bandwidth