



Ostfalia - Hochschule für angewandte Wissenschaften
Fakultät Informatik
Studiengang Informatik

Master-Arbeit

Reactive Programming mit Quarkus

eingereicht bei	Prof. Dr. B. Müller	
von	Erik Simonsen	70455429

Wolfenbüttel, den 31. Mai 2021

Zusammenfassung

Ehrenwörtliche Erklärung

Hiermit erkläre ich ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte fremde Hilfe angefertigt habe, andere als die angegebenen Quellen nicht benutzt und die benutzten Quellen wörtlich oder die inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wolfenbüttel, den 31. Mai 2021

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Aufbau	1
1.3	HTTP-basiertes Streaming	1
1.4	WebRTC	1
2	Vergleich HTTP-basiert und WebRTC	3
2.1	HTTP-basiertes Streaming	3
2.2	WebRTC	4
2.3	Fazit	4
3	System	5
3.1	WebSockets	5
3.2	SDP (Session Description Protocol)	5
3.3	ICE (Interactive Connectivity Establishment)	5
3.4	STUN und TURN Server	7
3.5	Signaling	7
4	Implementierung	9
4.1	getUserMedia	9
4.2	Signaling	9
4.3	UI	11
4.4	WebComponent	12
5	Fazit	13

Abbildungsverzeichnis

1	Ausschnitt einer SDP-Nachricht zwischen zwei <i>peers</i> auf <i>localhost</i>	6
2	WebRTC Signaling Ablauf, (Satanas 2020)	8
3	Codeausschnitt CreateOffer()-Methode	9
4	Codeausschnitt Datenaustausch	11
5	Codeausschnitt Custom <video-conference>-Element	12

1 Einleitung

1.1 Motivation

1.2 Aufbau

1.3 HTTP-basiertes Streaming

Beim HTTP-Streaming werden die darzustellenden Media-Inhalte über HTTP durch einen Webserver zur Verfügung gestellt. Dabei werden die Inhalte in eine Vielzahl an kleinen Datei-Segmenten zerlegt, welche jeweils einen Teil der Gesamtdauer enthalten. Die Segmente werden mehrfach, in unterschiedlichen Bitraten kodiert, auf dem Server hinterlegt. Dadurch kann der Client jederzeit das Segment auswählen das mit der derzeitigen Bandbreite, innerhalb der gegebenen Zeitgrenzen, vollständig übertragen werden kann, um somit ein sog. Buffering¹ bei dynamischen Netzwerkbedingungen weitestgehend zu verhindern. Diese Technik wird als *adaptive bitrate streaming* bezeichnet(Sodagar 2011)(Sodagar 2012). MPEG-Dash ist der erste internationale Standard im Bereich des *bitrate adaptive streaming*(MPEG 2011).

Informationen über die Segmente wie Dauer, URI, Reihenfolge und Bitraten werden in einer *media presentation description*, *MDP* festgehalten und zu Beginn dem Client übermittelt.(R. Pantos 2020, Seite 3) Dieser kann dann über die URI die Segmente nacheinander vom Server abrufen. Das Format und die Struktur der *MDP* sowie die genauen Implementierungsdetails (Segmentierung, Dateiformate, Videokompressionsverfahren) unterscheiden sich bei den populären HTTP-basierten Streaming-Protokollen, so wird die MPD bei HLS als Media Playlist bezeichnet(R. Pantos 2020). So gibt es unter Anderem:

- Dynamic Adaptive Streaming over HTTP (MPEG-DASH) von MPEG
- HTTP Live Streaming [HLS] von Apple
- Microsoft Smooth Streaming (MSS) von Microsoft

1.4 WebRTC

Bei WebRTC(Web Real-Time Communication) handelt es sich um einen offenen Standard, der vom World Wide Web Consortium (W3C) standardisiert wird. Maßgebliche Unterstützung und Entwicklungsarbeit erfolgt dabei von großen Browser-Herstellern wie Google Inc, Mozilla Foundation und Opera Software(Jennings 2019, Siehe Editors). Der Standard umfasst mehrere Programmierschnittstellen und Kommunikationsprotokolle, welche es Browsern ermöglichen auf bestimmte Hardware als Medienquelle zuzugreifen, und dessen Video-, Sprach- oder generische Daten mit Browsern auf anderen Rechnern (*peers*) in Echtzeit bidirektional auszutauschen. Bevor der Datenaustausch erfolgen kann, muss allerdings zuerst eine Verbindung zwischen zwei (oder mehreren) *peers* über einen zentralen Server(*signaling-server*) hergestellt werden ². Die Technologien, welche WebRTC

¹Ein Stoppen der Wiedergabe bis die nächsten Inhalte geladen sind

²Dieser Vorgang wird als *signaling* bezeichnet und ist nicht Teil des WebRTC-Standards (WebRTC 2020b)

umfasst, sind in allen populären Browsern als Javascript-APIs implementiert (WebRTC 2020a, Abschnitt *Signaling*). Die wesentlichen Komponenten sind:

- *getUserMedia* Gewährt Zugriff auf Audio- und Videoquellen (Webcam, Mikrofon) (Jennings 2019, Abschnitt 9)
- *RTCPeerConnection* Ermöglicht Austausch der Daten zwischen *peers* (Jennings 2019, Abschnitt 4.4). Kümmt sich um die *peer-to-peer* Kommunikation, Signalverarbeitung, Kodierung- und Dekodierung, sowie Sicherheit der Kommunikation (Verschlüsselung der Daten)
- *RTCDataChannel* Erlaubt die bidirektionale Kommunikation von beliebigen Daten zwischen *peers* mit sehr niedriger Latenz (Jennings 2019, Abschnitt 6).

2 Vergleich HTTP-basiert und WebRTC

In diesem Kapitel werden der WebRTC-Ansatz und der Ansatz eines HTTP-basierten Streaming-Protokolls bezüglich des Anforderungskontext eines web-basierten Konferenzsystems miteinander verglichen. Dabei ist es hilfreich, die essentiellen technischen Anforderungen an das genannte Konferenzsystem zu definieren:

- Sehr geringe Latenz -> Übertragung möglichst nahe an Echtzeit
- Wiedergabequalität als Tradeoff
- Serverlast möglichst gering, besonders bei hohen Teilnehmerzahlen
- Herstellerunabhängigkeit durch offene Standards

2.1 HTTP-basiertes Streaming

Durch die internationale Standardisierung von MPEG-DASH ist die Anforderung der Herstellerunabhängigkeit erfüllt, wobei die Auswahl einer geeigneten Implementation hierbei nicht betrachtet wird. Problematisch wird es bei der Nutzung dieses Ansatzes für eine Webkonferenz allerdings bei der Latenz und der Skalierbarkeit. Wie beim gängigen Live-Streaming, wobei es sich um eine 1:n Beziehung des Streamers zu seinen Zuschauern handelt, muss von der Quelle (z.B. die Webcam) zuerst, mithilfe eines Encoders, ein H.264-encodiertes Video³ erstellt werden. Dieses wird an den Webserver übermittelt und segmentiert, also mehrfach, in unterschiedlichen Bitraten kodiert, auf dem Server hinterlegt. Selbst bei dynamischen Netzwerkbedingungen, kann dadurch in der Regel eine flüssige Wiedergabe durch rechtzeitiges Laden von Segmenten mit passender Bitrate erfolgen, ohne die Wiedergabequalität zu stark zu beeinträchtigen. Die passenden Segmente werden anschließend vom Webserver an die anderen Teilnehmer verteilt.

Dies bringt für diesen Anwendungszweck jedoch mehrere Probleme mit sich. Zum Einen entsteht durch die Encodierung, den Transport zum Webserver, und die Segmentierung ein erheblicher Rechenaufwand und eine, dadurch bedingte, deutlich spürbare Zeitverzögerung (*delay*). Dieser *delay* wird zusätzlich durch das *Buffering*, verstärkt. Für eine flüssige Wiedergabe ist ein kurzes Buffering notwendig, um die anfänglichen Mediensegmente zu laden (R. Pantos 2020, 3.2 Partial Segments). Zudem erfolgt die Übertragung per TCP, um die Vollständigkeit der Datenpakete, und somit eine hohe Wiedergabequalität, zu gewährleisten (R. Pantos 2020, 4.1 Definition of a Playlist). Da es sich bei einer Webkonferenz zudem um eine n:m Beziehung zwischen den Teilnehmern handelt könnten Webserver bei höheren Nutzerzahlen schnell das Performance-Bottleneck der Anwendung werden und somit schlecht skalieren. Dies würde daran liegen, dass die regelmäßige Segmentierung der encodierten Aufnahmevideos für jeden einzelnen Benutzer erfolgen müsste, was bei hohen Benutzer- und Konferenzzahlen einen immensen serverseitigen Rechenaufwand bedeuten würde.

³Die genutzte Codec ist nicht standardisiert und auch anders gewählt werden.

2.2 WebRTC

Bei WebRTC werden die Daten direkt zwischen den Clients übertragen, man spricht von einer Rechner-zu-Rechner (*Peer-to-Peer*) Kommunikation. Der Server ist hierbei nur für die Lokalisierung und Herstellung der Kommunikation (*Signaling*) zwischen den Clients zuständig und verwaltet eine Liste der aktiven Teilnehmer, informiert Sie über einen neuen Teilnehmer etc. Die Aufnahmequelle des Benutzers wird im Browser über die `getUserMedia()` API abgerufen und der daraus resultierende *MediaStream* kann direkt vom Browser in die benötigte Codec encodiert werden⁴(Docs o.D.[a]).

Durch die *RTCPeerConnection* API kann der encodierte *MediaStream* nun direkt an die anderen Clients geschickt werden. Da der Webserver, außer für das Signaling, nicht benötigt wird, ist die Serverlast sehr gering. Das Encodieren, Dekodieren und Verschlüsseln des *MediaStream*, sowie das Aufrechterhalten der Kommunikation wird komplett vom Browser der Nutzer erledigt, wodurch der Prozess verhältnismäßig viele Ressourcen des Rechners benötigt(Jennings 2019, 5. RTP Media API). Daher ist es durchaus möglich, dass bei vielen Clients Performanceprobleme bei leistungsschwachen Rechnern auftreten können. Aufgrund der direkten Kommunikation der Clients sind die Latenzzeiten sehr gering, und erfüllen die weiche Echtzeitanforderung.

Es kann jedoch zu Paketverlust kommen, da die Datenübertragung standardmäßig per UDP erfolgt und sich die Wiedergabequalität somit kurzzeitig verschlechtern kann(Jennings 2019, 14. Accessibility Considerations). Zudem kann die Wiedergabequalität serverseitig nicht beeinflusst werden, sondern ist abhängig von verfügbaren Bandbreite der Nutzer. Durch die n:m Beziehung bei Webkonferenzen skaliert auch der WebRTC Ansatz nur begrenzt. Da die Hauptlast vom Browser bewältigt wird, gibt es von den verschiedenen Browserherstellern Vorgaben, wieviele gleichzeitige aktive *RTCPeerConnections* ein Client haben darf.

2.3 Fazit

Basierend auf den genannten Aspekten kommt der Autor zu dem Schluss, dass ein Http-basierter Ansatz sich *nicht* für den Anwendungskontext einer Webkonferenz eignet. Zum Einen ist der bestehende Delay nicht geeignet für eine Webkonferenz, da diese echtzeitfähig sein muss, und zum Anderen ist die Last auf den/die Webserver sehr hoch, wenn die Medienquelle jedes Client segmentiert werden muss, besonders wenn die Nutzerzahl und Anzahl der Konferenzen hoch ist. Bitraten-Adaptives Streaming über HTTP eignet sich jedoch hervorragend für On-Demand Streaming und Live Streaming. Beim On-Demand Streaming liegen die Mediensegmente des Mediums bereits auf dem Server, und müssen nicht dynamisch generiert werden, und werden über ein CDN (Content Deliver Network) an die Clients verschickt. Beim Live-Streaming muss die Segmentierung zwar dynamisch durchgeführt werden, allerdings nur für die Medienquelle des Streamers und die Zuschauer rufen diese Segmente lediglich ab. Bei beiden Streaming-Bereichen herrscht also eine 1:n Beziehung zwischen Quelle und Benutzern und dementsprechend skaliert HTTP-Streaming hier auch besser als in einer Webkonferenz. Zudem ist eine hohe Wiedergabequalität bei den genannten Bereichen wichtiger als eine niedrige Latenz.

⁴Falls der Browser die Codec unterstützt

WebRTC eignet sich nach Meinung des Autors hingegen sehr gut. Durch die direkte Peer-To-Peer Kommunikation ist die Latenz sehr gering, und erfüllt das Echtzeitkriterium. Zudem ist die Serverlast sehr gering, da der rechenintensive (Transcoding und Verschlüsselung des MediaStreams) Teil komplett clientseitig ausgeführt wird und lediglich das Signaling serverseitig ausgeführt werden muss. Darüber hinaus ersparen die RTCPeerConnection-API und die getUserMedia-API einen erheblichen Teil komplexer Entwicklungsarbeiten, wie beispielsweise das Konvertieren von Codecs. Der mögliche, auftretende Paketverlust durch den Transport der Datenpakete durch UDP fällt wenig ins Gewicht, kann aber möglicherweise für Enterprise Software ein Problem darstellen.

3 System

3.1 WebSockets

Für das Signaling ist eine Kommunikation über mehrere Schritte hinweg notwendig. Damit diese ohne eine Vielzahl an HTTP-Anfragen und komplexer serverseitiger Vorkonfiguration realisiert werden können, wird client- und serverseitig die WebSocket-API (MDN 2020) genutzt⁵. Da WebRTC allerdings keinen Transport-Mechanismus für die Signaling-Informationen spezifiziert, ist es auch möglich die Informationen über XMLHttpRequests zu versenden (sog. Ajax-Requests). WebSockets ermöglichen die bidirektionale Client-Server Kommunikation. Dabei kann der Client dem Server Nachrichten senden und ereignisorientierte Antworten erhalten, ohne Antworten durch HTTP-Anfragen abzufragen (docs 2020f). Die Nachrichten werden dabei im JSON-Format ausgetauscht. Folgende Ereignisse können, sowohl client- als auch serverseitig, ausgelöst werden:

- OnOpen()
- OnMessage()
- OnClose()
- OnError()

3.2 SDP (Session Description Protocol)

SDP ist ein Standardformat um eine *peer-to-peer* Verbindung zu beschreiben und wird u.A. von WebRTC genutzt um Sitzungen von *peers* zu beschreiben. Es enthält Informationen zur verwendeten Codec, den IP-Adressen der *peers*, und zu Zeitabschnitten von Video- und Audiodaten (docs 2020d).

3.3 ICE (Interactive Connectivity Establishment)

ICE ist ein von WebRTC genutztes Framework um zwei Peers miteinander zu verbinden, unabhängig von der vorhandenen Netzwerk-Topologie. (IETF 2018) Das ICE-Protokoll ermöglicht das finden und etablieren einer Verbindung zwischen zwei *peers*, selbst wenn

⁵Serverseitig wird die WebSocket API von Java genutzt (Oracle 2020)

```

1  "v=0
2  o=- 1576712127687105939 2 IN IP4 127.0.0.1
3  s=-
4  t=0 0
5  a=group:BUNDLE 0 1
6  a=msid-semantic: WMS bDMDR2TAYjeitQFA2DwEXlZyO8rwb1FBWUZ
7  m=audio 9 UDP/TLS/RTP/SAVPF 111 103 104 9 0 8 106 105 13 110 112 113 126
8  c=IN IP4 0.0.0.0
9  a=rtcp:9 IN IP4 0.0.0.0
10 a=ice-ufrag:5mC0
11 a=ice-pwd:v4HJb4KKPo7Udb0Noe2PUCf1
12 a=ice-options:trickle
13 a=fingerprint:sha-256 87:C2:8D:8E:30:7C:88:73:AD:40:32:24:B6:66:68:39:57:
14 a=setup:active
15 a=mid:0
16 a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
17 a=extmap:2 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
18 a=extmap:3 http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-ex
19 a=extmap:4 urn:ietf:params:rtp-hdrext:sdes:mid
20 a=extmap:5 urn:ietf:params:rtp-hdrext:sdes:rtp-stream-id
21 a=extmap:6 urn:ietf:params:rtp-hdrext:sdes:repaired-rtp-stream-id
22 a=sendrecv
23 a=msid:bDMDR2TAYjeitQFA2DwEXlZyO8rwb1FBWUZ 45296bed-c049-4375-b7b4-15326
24 a=rtcp-mux
25 a=rtpmap:111 opus/48000/2
26 a=rtcp-fb:111 transport-cc
27 a=fmtp:111 minptime=10;useinbandfec=1

```

Abbildung 1: Ausschnitt einer SDP-Nachricht zwischen zwei *peers* auf *localhost*

beide eine Netzwerkadressübersetzung(NAT) nutzen, um eine öffentliche IP Adresse mit anderen Geräten in ihrem Netzwerk zu teilen. Dafür kann im Konstruktor eines `RTCPeerConnection`-Objektes ein STUN und/-oder TURN Server angegeben werden.

Der Algorithmus des Frameworks versucht dabei immer den Pfad mit der niedrigsten Latenz zu nutzen um die beiden *peers* zu verbinden. Dabei probiert er die folgenden Optionen in der genannten Reihenfolge:

1. Direkte UDP Verbindung (in diesem Fall wird ein STUN Server genutzt um die IP-Adresse des *peers* innerhalb des Netzwerkes zu finden)
2. Direkte TCP Verbindung, über den HTTP-Port
3. Direkte TCP Verbindung, über den HTTPS-Port

4. Indirekte Verbindung über einen TURN Server (wenn eine direkte Verbindung fehlschlägt, bspw. wenn ein *peer* hinter einer Firewall liegt, die die NAT-Traversierung blockiert)(docs 2020c)

3.4 STUN und TURN Server

STUN (Session Traversal Utilities for NAT) ist ein Hilfsprotokoll um Daten durch ein NAT (Network Address Translator) an das Ziel zu übermitteln. Es gibt die IP-Adresse, den Port, und den Verbindungsstatus des Computers hinter dem NAT zurück(docs 2019a) und ermöglicht somit eine direkte Kommunikation der beiden *peers*.

TURN (Traversal Using Relays around NAT) ist ein Protokoll das einem Computer - hinter einem NAT oder einer Firewall - ermöglicht Daten zu senden und empfangen. Dabei wird der gesamte Traffic der beiden *peers* während den *gesamten* Kommunikationszeitraums über den TURN-Server an den jeweils anderen *peer* weitergeleitet. Der TURN-Server muss dabei in der Regel öffentlich zugänglich sein. TURN-Server werden dann genutzt, wenn eine direkte Kommunikation der *peers* durch eine sehr restriktive Firewall nicht möglich ist⁶(docs 2019b).

3.5 Signaling

Ein Signaling-Server fungiert als Mittelsmann und ermöglicht es zwei *peers* sich zu finden und eine Verbindung aufzubauen, indem er die Signaling-Daten jeweils an den entsprechenden *peer* weiterleitet. Dabei muss der Server den Inhalt der Signaling-Nachrichten nicht verstehen ⁷ können, solange der jeweilige *peer* die Nachricht erhält und an sein ICE-System (siehe Abschnitt *ICE*) weitergibt. WebRTC nutzt für die Inhalte der Signaling-Nachrichten das *Session Description Protocol (SDP)* und für die Kommunikation mit dem Signaling-Server werden *WebSockets* empfohlen.

Zu Beginn des Signaling Prozesses initialisiert Peer A ein *RTCPeerConnection*-Objekt und erstellt ein Offer mithilfe der *CreateOffer()*-Methode. Das Offer enthält seine Session-Description im SDP-Format und wird über den Signaling-Server an Peer B geschickt, welcher auch ein *RTCPeerConnection* erstellt. Dieser verarbeitet das Offer und dessen Inhalt und reagiert mit einer Answer, die wiederum die Session Description von Peer B enthält.

Nachdem Peer A die Answer verarbeitet hat, haben beide *peers* das jeweilige *RTCPeerConnection*-Objekt mit ihrer und der Session-Description des anderen Peers gefüllt, sowie den Audio- und Videodaten der jeweiligen Medienquelle als *MediaStream*-Objekt.

Beide *peers* kennen nun die IP-Adresse des jeweils anderen und die zu nutzenden Codecs, wissen noch allerdings nicht wie die Mediendaten übertragen werden sollen. Um dies zu ermitteln, wird das ICE-Framework von WebRTC genutzt.

Die *peers* tauschen ICE-Kandidaten aus um die aktuelle Verbindung zu verhandeln.

Jeder ICE-Kandidat beschreibt eine Methode die der sendende *peer* zur Kommunikation nutzen kann.

⁶Bei permanent hohem Traffic der *peers* kann die Nutzung eines externen TURN-Servers ein Performance-Bottleneck darstellen.

⁷Der Inhalt der Signaling-Nachrichten kann daher als Black-Box gesehen werden.

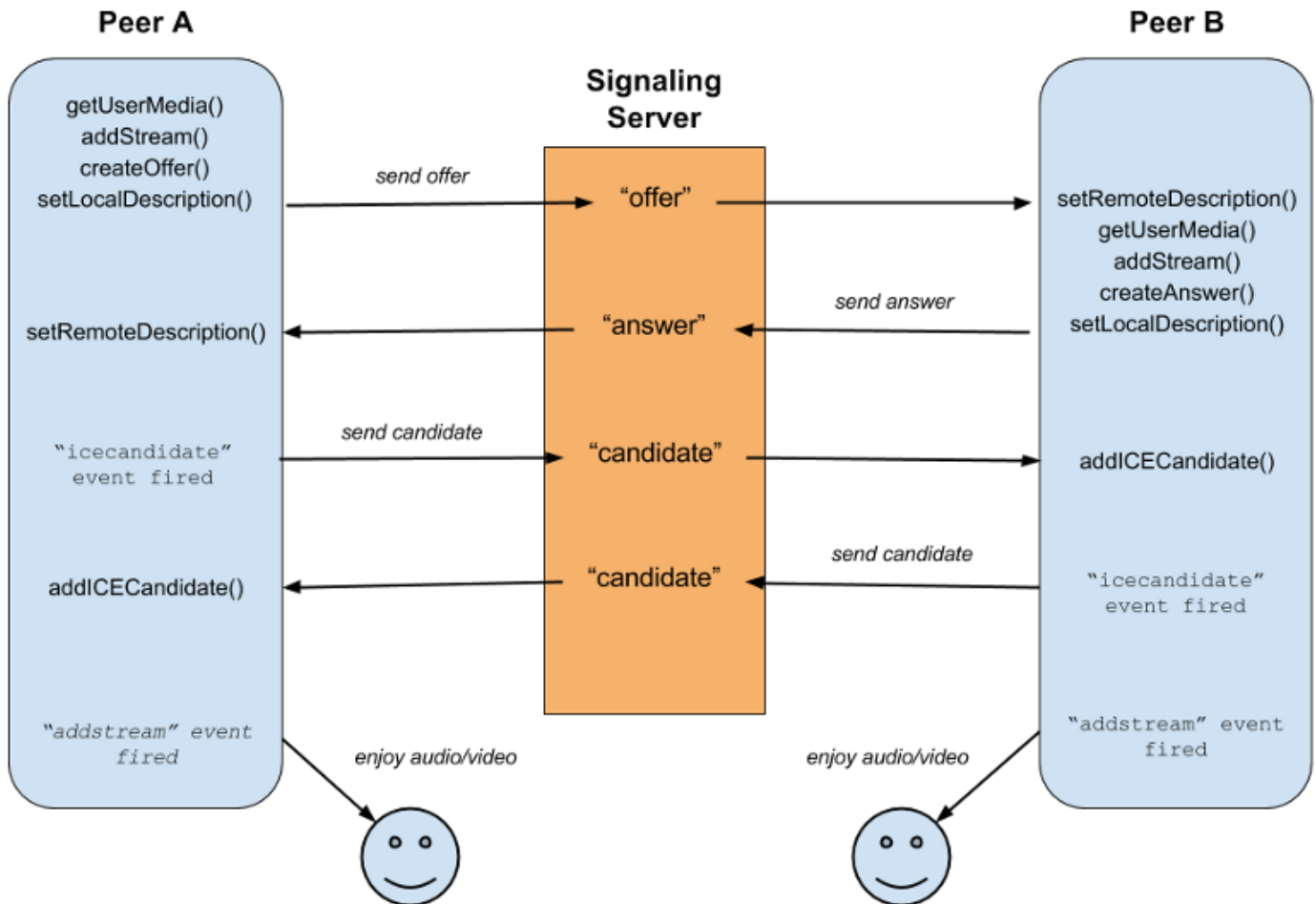


Abbildung 2: WebRTC Signaling Ablauf, (Satanas 2020)

Beide *peers* senden Kandidaten in der Reihenfolge, in der sie entdeckt werden, und senden solange bis sie keine Vorschläge mehr haben - selbst wenn die Mediendaten bereits übertragen werden. Jede ICE-Nachricht schlägt ein Kommunikationsprotokoll (TCP oder UDP), IP-Adresse, Port und andere Informationen vor, die benötigt werden um eine Verbindung aufzubauen. Dies beinhaltet auch NAT .

Sobald sie sich auf einen beidseitig-kompatiblen Kandidaten geeinigt haben, wird das SDP des Kandidaten von jedem Peer genutzt um eine Verbindung aufzubauen, durch welche die Mediendaten ausgetauscht werden.

Falls Sie sich später auf einen besseren Kandidaten (normalerweise mit besserer Performance) einigen, ändert sich das Format des Datenstroms.

(docs 2020e)

4 Implementierung

4.1 getUserMedia

Zu Beginn der Anwendung wird die Wiedergabequelle des Clients in Form eines `MediaStream`-Objektes abgerufen. Die Umsetzung ist durch die `getUserMedia-API()` sehr einfach und beschränkt sich auf wenige Zeilen-Code.

Hierbei wird der Benutzer aufgefordert dem Browser Zugriff auf eine Medienquelle zu gewähren, die einen `MediaStream` produziert der die angegebenen Medientypen (Video und Audio) als Spuren (`MediaStreamTrack`) enthält. Anzumerken ist hierbei das die Methode `getUserMedia()` asynchron ist. Für die Nutzung der API außerhalb der lokalen Entwicklung (localhost) muss die Anwendung zudem über HTTPS geladen werden (*secure context*) (Docs o.D.[b]).

Nachdem dies erfolgt ist beginnt der Signaling-Prozess.

4.2 Signaling

Für den Austausch der notwendigen Signaling-Nachrichten werden client- und serverseitig WebSockets verwendet. Clientseitig wird dafür die WebSocket-API genutzt und einfach über `new WebSocket()` das Objekt erstellt. Serverseitig wird die Java-API für WebSocket (JSR 356) genutzt und mit der `@ServerEndpoint` Annotation ein Endpunkt erstellt.

Da es sich bei der entwickelten Anwendung nicht nur um einen Videochat zwischen zwei *peers* handelt, sondern jeder *peer* mit einer Vielzahl an anderen *peers* (n:m Beziehung) kommunizieren muss, müssen client- und serverseitig die Teilnehmer mitgeführt werden. Im Browser wird dafür ein Array an `RTCPeerConnection`-Objekten verwaltet und auf dem Signaling-Server eine Menge an `HttpSession`-Objekten.

Sobald nun ein neuer Client ein WebSocket-Verbindung zu dem Server-Endpunkt aufbaut wird beiderseits das `onOpen()`-Event ausgelöst. Der Server fügt ein neues `HttpSession`-Objekt hinzu, und informiert alle bestehenden Clients über die neue Verbindung, sowie den neuen Client über alle bestehenden Clients.

Anschließend legt jeder bestehende Client ein neues `RTCPeerConnection`-Objekt an und erstellt ein Offer über `createOffer()`, also eine Nachricht mit der eigenen Session-Description und schickt diese an den Signaling-Server. Der neue Client muss für jeden bestehenden Client eine neue `RTCPeerConnection` erstellen und auf die Offers der anderen Clients warten. Nun wird der Kommunikationsaufbau analog zu Abbildung 2 ausgeführt.

```
createOffer(senderId) {
    this.peerConnections[senderId].createOffer().then(function (sessionDescription) {
        this.peerConnections[senderId].setLocalDescription(sessionDescription).then(function () {
            this.sendToServer( msg: {
                event: "offer",
                senderSessionId: this.localSessionId,
                targetSessionId: senderId,
                data: sessionDescription
            });
        });
    });
}
```

Abbildung 3: Codeausschnitt CreateOffer()-Methode

Bei jeder Nachricht während des Signaling-Prozesses muss immer die Session-ID des Senders und des Empfängers angegeben werden, damit der Server die Weiterleitung an den entsprechenden Client abwickeln kann, und der Empfänger die Nachricht dem entsprechenden `RTCPeerConnection`-Objekt des Senders zuordnen kann. Zusätzlich muss immer ein Nachrichten-Typ übermittelt werden, damit der Empfänger weiß wie er das SDP der Nachricht verarbeiten muss und wie er antworten muss. Dafür muss im clientseitigen `onMessage` Event des `WebSockets` eine Differenzierung, je nach empfangenen Nachrichten Typ erfolgen.

4.3 UI

Nachdem sich beide *peers* auf einen ICE-Kandidaten geeinigt haben, und somit die Medien-Daten austauschen können, wird das *onTrack*-Event der *peers* ausgelöst⁸. Die *MediaStreamTrack*-Objekte werden den *RTCPeerConnection*-Objekten bereits nach deren Initialisierung zugewiesen, sofern dann schon ein *MediaStream*-Objekt vorhanden ist.

```
this.peerConnections[roomParticipant.sessionId] = new RTCPeerConnection(this.servers);
this.peerConnections[roomParticipant.sessionId].onicecandidate = function (event) {
  if (event.candidate !== null) {
    this.sendToServer( msg: {
      event: "candidate",
      senderSessionId: this.localSessionId,
      targetSessionId: roomParticipant.sessionId,
      data: event.candidate
    });
  }
}.bind(this);
this.peerConnections[roomParticipant.sessionId].ontrack = function (event) {
  this.gotRemoteStream(roomParticipant.sessionId, roomParticipant.userName, event);
}.bind(this);
for (const track of this.localStream.getTracks()) {
  this.peerConnections[roomParticipant.sessionId].addTrack(track, this.localStream);
}
```

Abbildung 4: Codeausschnitt Datenaustausch

Im Event-Handler des *onTrack* -Events kann nun auf den übertragenen *MediaStream*, und dessen *Tracks*, des anderen *peers* zugegriffen werden. Für die Oberfläche dieser Anwendung wird im Event-Handler eine Funktion aufgerufen in der für jeden neuen *MediaStream* ein *HTMLVideo*-Element erstellt, und mit der *append()*-Methode in den DOM eingefügt, wird. Da das *HTMLVideo*-Element das *HTMLMedia-Element*-Interface implementiert kann als Wiedergabe-Quelle ein *MediaStream*-Objekt gesetzt werden(docs 2020a).

⁸Die *addTrack()*-Methode löst das *onTrack*-Event des *RTCPeerConnection*-Objektes auf der Remote Seite aus, nicht bei dem lokalen *RTCPeerConnection*-Objekt

4.4 WebComponent

Um eine möglichst hohe Wiederverwendbarkeit und leichte Anpassung für andere Projekte zu ermöglichen, wurde die clientseitige Logik als sogenannte WebComponent gekapselt. Dabei handelt es sich um ein benutzerdefiniertes HTML-Element, welches mit einer Javascript-Klasse verknüpft ist und die dort definierte Funktionalität über die zu implementierende Methode *connectedCallback()* ausführt sobald es vom DOM mit dem Dokument verbunden wird. Die Verknüpfung des `<video-conference>`-Elementes mit der VideoConference-Klasse wird durch die `CustomElementRegistry.define()`-Methode registriert. Die VideoConference-Klasse muss zudem von der `HTMLElement`-Klasse erben. Wenn die Logik nun in einem anderen Projekt für Videokonferenzen verwendet wird, kann der Endpunkt des Signaling-Servers über die HTML-Attribute *socketHostname* und *socketPathname* modifiziert werden, ohne den Javascript-Code anpassen zu müssen. Die Existenz und der Wert dieser Attribute wird in der `connectedCallback()`-Methode abgefragt.

```
<video-conference socketPathname="/video-conference/signaling" template="video-conference-default">
  <span slot="video-conference headline">Video Konferenz</span>
</video-conference>
```

Abbildung 5: Codeausschnitt Custom `<video-conference>`-Element

Um zudem die Element-Struktur, das Styling und das Verhalten vom restlichen Code der Anwendung abzugrenzen wird die Shadow DOM API genutzt. Dabei handelt es sich um einen versteckten separaten DOM der an ein Element angehängt wird und nicht von den Styles und definierten Verhalten des Standard-DOMs betroffen (docs 2020b).

5 Fazit

Bezug auf in Einleitung angegebene Ziele

Literatur

- docs, MDN web (2019a). URL: <https://developer.mozilla.org/en-US/docs/Glossary/STUN> (besucht am 23.07.2020).
- (2019b). URL: <https://developer.mozilla.org/en-US/docs/Glossary/TURN> (besucht am 23.07.2020).
- (2020a). URL: <https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/srcObject> (besucht am 26.07.2020).
- (2020b). URL: https://developer.mozilla.org/de/docs/Web/Web_Components/Using_shadow_DOM (besucht am 26.07.2020).
- (2020c). *ICE*. URL: <https://developer.mozilla.org/en-US/docs/Glossary/ICE> (besucht am 23.07.2020).
- (2020d). *SDP*. URL: <https://developer.mozilla.org/en-US/docs/Glossary/SDP> (besucht am 23.07.2020).
- (2020e). *Signaling and video calling*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling (besucht am 22.07.2020).
- (2020f). *WebSockets*. URL: <https://developer.mozilla.org/de/docs/WebSockets> (besucht am 21.07.2020).
- Docs, MDN Web (o.D.[a]). *Codecs used by WebRTC*. URL: https://developer.mozilla.org/en-US/docs/Web/Media/Formats/WebRTC_codecs.
- (o.D.[b]). *getUserMedia - Security*. URL: <https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia#Security>.
- IETF (2018). *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. URL: <https://tools.ietf.org/html/rfc8445> (besucht am 23.07.2020).
- Jennings Boström, Bruaroey (2019). *WebRTC 1.0: Real-time Communication Between Browsers*. URL: <https://www.w3.org/TR/webrtc> (besucht am 13.12.2019).
- MDN (2020). *HTML Living Standard - 9.3 Web Sockets*. URL: <https://html.spec.whatwg.org/multipage/web-sockets.html> (besucht am 21.07.2020).
- MPEG (2011). *MPEG ratifies its draft standard for DASH*. URL: https://web.archive.org/web/20120820233136/http://mpeg.chiariglione.org/meetings/geneva11-1/geneva_press.htm (besucht am 18.08.2020).
- Oracle (2020). *Annotated Endpoints*. URL: <https://javaee.github.io/tutorial/websocket004.html> (besucht am 21.07.2020).
- R. Pantos, Apple Inc. (2020). *HTTP Live Streaming 2nd Edition*. URL: <https://tools.ietf.org/html/draft-pantos-hls-rfc8216bis-07.html> (besucht am 30.04.2020).
- Satanas (2020). *simple-signaling-server*. URL: <https://github.com/satanas/simple-signaling-server> (besucht am 22.07.2020).

- Sodagar (2012). *White paper on MPEG-DASH Standard*. Erster Downloadlink "White Paper on MPEG Dash". URL: <https://mpeg.chiariglione.org/standards/mpeg-dash> (besucht am 14.07.2020).
- (2011). URL: <https://mpeg.chiariglione.org/standards/mpeg-dash/media-presentation-description-and-segment-formats> (besucht am 03/2011).
- WebRTC (2020a). URL: <https://webrtc.org> (besucht am 06.07.2020).
- (2020b). *Getting started with peer connections*. URL: <https://webrtc.org/getting-started/peer-connections> (besucht am 06.07.2020).