BACHELORPROEF

Het implementeren van remote access in een browser.

Hoe wordt de mogelijkheid om meerdere personen gelijktijdig controle te geven over een extern scherm in een browser geïmplementeerd?

Bachelor	Toegepaste Informatica
Keuzetraject	Computer & Cyber Crime Professional
Academiejaar	2020 - 2021
Student	Sarah Polfliet
Interne begeleider	Ann Audenaert (Howest)
Externe promotor	Davy De Coster (Ocular)

howest.be

BACHELORPROEF

Het implementeren van remote access in een browser.

Hoe wordt de mogelijkheid om meerdere personen gelijktijdig controle te geven over een extern scherm in een browser geïmplementeerd?

Bachelor	Toegepaste Informatica
Keuzetraject	Computer & Cyber Crime Professional
Academiejaar	2020 - 2021
Student	Sarah Polfliet
Interne begeleider	Ann Audenaert (Howest)
Externe promotor	Davy De Coster (Ocular)

howest.be

Toelating tot bruikleen

Woord vooraf

De bachelorproef die u zo dadelijk zal lezen, gaat over het implementeren van remote access control in een browser. Dit onderwerp kwam er na een researchopdracht bij Ocular, het bedrijf waar ik stage heb gelopen.

Graag wil ik meneer Vanden Avenne Nicolas, mevrouw Mattheeuws Sofie en meneer Denys Simon, zaakvoerders van Ocular, bedanken om mijn stage bij Ocular te mogen doen en om mij dit onderwerp te laten onderzoeken.

Ik zou meneer De Coster Davy, backend developer en stagementor bij Ocular, willen bedanken voor alle tips en feedback tijdens mijn stage maar ook bij het opbouwen van deze bachelorproef.

Ook wil ik mevrouw Audenaert Ann, lector en stagebegeleidster bij Howest, bedanken voor de feedback op deze bachelorproef.

Tenslotte wil ik mevrouw Terryn Heidi, lector bij Howest, bedanken voor de tips rond het schrijven van een bachelorproef.

Sarah Polfliet

Huldenberg, 8 juni 2021

Samenvatting

Deze bachelorproef gaat over de implementatie van remote access control in een browser. Er zijn verschillende belangrijke aspecten hierin. Eerst en vooral is het de bedoeling om een manier te vinden om een scherm te delen. Dit wordt gedaan door middel van het WebRTC-protocol. Hiermee is het mogelijk om zogenaamde mediastreams uit te wisselen. In de bachelorproef wordt de werking van WebRTC in detail besproken.

De remote access control is een ander groot aspect. Dit beschrijft hoe we een remote scherm kunnen bedienen. Daarbij moeten ook mogelijke security problemen behandeld worden. Er moet rekening gehouden worden met het feit dat er ook risico's verbonden zijn aan remote control. Het is bijvoorbeeld niet de bedoeling dat een hacker via remote control aan het bedrijfsnetwerk kan en hier schade kan toebrengen.

Iframes zijn ook belangrijk omdat deze vaak worden gebruikt in applicaties. Daarnaast wil ik aantonen dat de remote access control ook hierbij werkt. Een iframe is een onderdeel van een HTML-element waarmee je documenten, video's en interactieve media kan insluiten in een pagina. Wanneer de inhoud van de iframe van een andere bron komt, kan dit zorgen voor security risico's. De meeste remote access control applicaties gaan de iframe daarom blokkeren.

Tenslotte wordt dit geïmplementeerd in een proof of concept. De technologieën die hiervoor worden gebruikt zijn Node.js, Express.js, EJS, Socket.io, PeerJS, RobotJS en RobotJS-browser. Deze worden dus ook besproken. Op het einde is er een handleiding om zelf aan de slag te gaan. Er worden ook nog enkele tips gegeven om nog eens te bekijken wanneer deze functionaliteit in een eigen applicatie wordt geïmplementeerd.

Sleutelwoorden:
Remote access control, iframe, WebRTC, Node.js, Express.js, EJS, Socket.io, PeerJS, RobotJS, RobotJS-browser

Abstract

This bachelor's thesis is about implementing remote access control in a browser. There are several important aspects in this. First and foremost, the idea is to find a way to share a screen. This is done here by means of the WebRTC protocol. With this it is possible to exchange so-called media streams. The bachelor's thesis will discuss the operation of WebRTC in detail.

The remote access control is another major aspect. This describes how we can control a remote screen. Thereby possible security problems have to be addressed as well. It must be taken into account that there are also risks associated with remote control. For example, it shouldn't be possible for a hacker to access the company network via remote control and cause damage.

Iframes are also important because they are often used in applications. I also want to demonstrate that the remote access control is possible in an iframe. An iframe is part of an HTML element that allows you to embed documents, videos and interactive media in a page. When the content of the iframe comes from another source, it can create security risks. Most remote access control applications will therefore block the iframe.

Finally, all this is implemented in a proof of concept. The technologies used for this are Node.js, Express.js, EJS, Socket.io, PeerJS, RobotJS and RobotJS browser. So these technologies are also discussed. At the end there is a tutorial to get started yourself. There are also some tips to revisit when implementing this in your own application.

Keywords:
Remote access control, iframe, WebRTC, Node.js, Express.js, EJS, Socket.io, PeerJS, RobotJS, RobotJS-browser

Lijst met figuren

Lijst met tabellen

Verklarende woordenlijst

Remote	Vanop afstand. [1]
Codec	Soft- of hardware die toelaat data te coderen/decoderen of te comprimeren/decomprimeren. [2]
Connection state tracking	De mogelijkheid om statusinformatie over een verbinding in geheugentabellen bij te houden. [3]
Congestion control	Een manier om de hoeveelheid gegevens die in een netwerk binnenkomen te monitoren. [4]
Stream multiplexing	Hiermee is het mogelijk om meerdere onafhankelijke logische streams een gemeenschappelijk onderliggend transportmedium te laten delen. [5]
Flow control	Het beheren van de snelheid van gegevensoverdracht tussen verzender en ontvanger, om te voorkomen dat een snelle verzender een trage ontvanger overweldigd. [6]
Keepalive	Een bericht dat door het ene apparaat naar het andere wordt verzonden om te controleren of de link tussen de twee werkt of om te voorkomen dat de link wordt verbroken. [7]
NAT traversal	Een computernetwerktechniek voor het tot stand brengen en onderhouden van IP-verbindingen tussen gateways die netwerkadresvertaling implementeren. [8]
Head-of-line blokkering	Een prestatiebeperkend fenomeen dat optreedt wanneer een reeks pakketten wordt opgehouden door het eerste pakket. [9]

Inhoudsopgave

1 Inleiding

1.1 Stage bij Ocular

Ocular is een bedrijf in Zwevezele dat zich vooral inzet in het bedenken en realiseren van belevingen en ervaringen die zowel bezoekers, gasten en klanten aanspreken. Dit doen ze op verschillende manieren.

Met Xperify kunnen bedrijven gebruikmaken van een virtuele showroom waarbij het bedrijf klanten persoonlijk kan ontvangen en doorheen hun aanbod kan gidsen. Zo kunnen klanten kennismaken met het bedrijf in een unieke 3D-omgeving.

Naast Xperify worden er nog digitale showrooms gebouwd waarbij gebruikers op een interactieve manier een bedrijf of concept leren kennen. En er zijn nog talloze andere projecten waaraan gewerkt wordt.

Tijdens mijn stage kreeg ik de kans om aan verschillende projecten mee te werken. Daarnaast kon ik een beter zicht krijgen op hoe het bedrijfsleven er uitziet. Ik leerde er Vue.js en Laravel beter kennen en kreeg ook te zien wat de andere developers maakten.

1.2 Remote control in een browser

Ik kreeg de opdracht om een applicatie te zoeken die remote control aanbiedt én die kan gebruikt worden in de eigen applicatie in de browser. Al snel werd duidelijk dat bestaande applicaties deze combinatie niet aanbieden. Er moest dus zelf gezocht worden naar een manier om remote access control te implementeren in de eigen applicatie in de browser. Daarbij komt nog eens de vraag om dit ook te laten werken met iframes.

2 Onderzoeksvraag

Uit de researchopdracht die op de vorige pagina besproken werd, kwam de volgende onderzoeksvraag:

“Hoe wordt de mogelijkheid om meerdere personen gelijktijdig controle te geven over een extern scherm in een browser geïmplementeerd?”

In deze bachelorproef wordt deze vraag in verschillende onderdelen verdeeld. Ik ga op zoek naar de werking van het WebRTC-protocol, wat belangrijk is voor het delen van media streams. Daarna omschrijf ik wat remote access control exact is en wat de gevaren ervan zijn. Ook de werking van iframes wordt bestudeerd.

Dit wordt allemaal praktisch uitgewerkt in een proof of concept. Hiervoor worden eerst de gebruikte technologieën besproken en daarna leg ik gedetailleerd uit wat ik allemaal deed om remote access in de browser te implementeren.

3 Onderzoek

3.1 WebRTC

WebRTC of Web Real-Time Communication is een collectie van standaarden, protocollen en JavaScript API's. Deze combinatie maakt peer-to-peer audio, video en data sharing tussen browsers mogelijk.

De meeste complexiteit wordt onderverdeeld in 3 API's:

- MediaStream voor het verkrijgen van audio- en videostreams.
- RTCPeerConnection is verantwoordelijk voor het beheer van de volledige levenscyclus van elke peer-to-peer verbinding.
- RTCDataChannel voor de communicatie van willekeurige applicatie data.

De architectuur van WebRTC wordt bepaald door 2 organisaties:

- Het World Wide Web Consortium (W3C) bepaalt de browser API's.
- De Internet Engineering Task Force (IETF) bepaalt de protocollen, data formaten, security… IETF is verantwoordelijk voor alle aspecten die nodig zijn om peer-to-peer communicatie in de browser mogelijk te maken.

3.1.1 Real-Time Communicatie

Aangezien real-time communicatie tijdgevoelig is, moet een applicatie kunnen omgaan met periodiek pakketverlies. De codecs kan kleine ontbrekende pakketten invullen waardoor er een minimale impact is. De applicatie moet zelf de verloren of vertraagde pakketten kunnen herstellen.

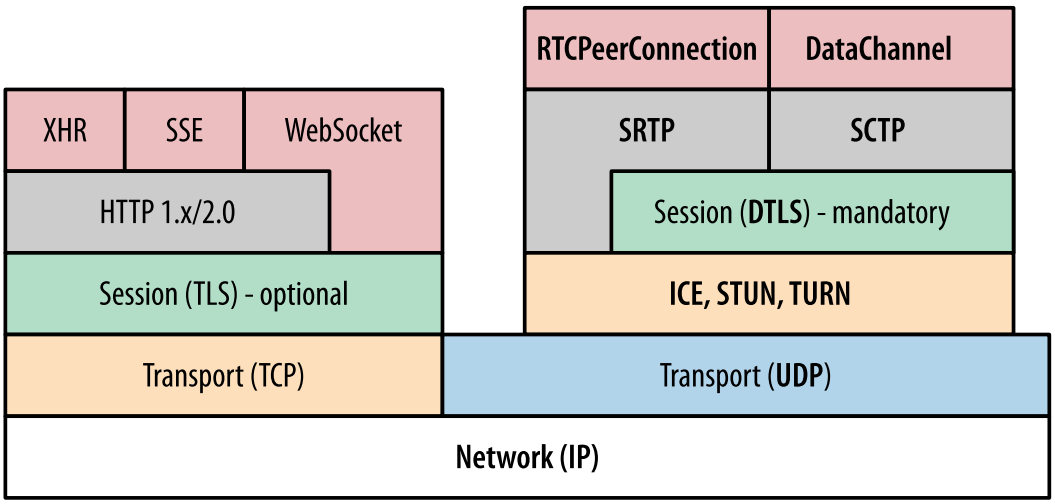
Een snelle communicatie is een zeer belangrijk gegeven voor WebRTC. Om dit te kunnen garanderen, wordt met het UDP-protocol gewerkt. UDP zorgt voor deze snelle verbinding door services die TCP aanbiedt niet te doen. Services die niet worden aangeboden zijn hieronder opgesomd:

- Geen garantie dat pakketten worden afgeleverd.
- Geen garantie dat pakketten in de juiste volgorde worden afgeleverd.
- Geen connection state tracking.
- Geen congestion control.

UDP is de basis voor real-time communicatie in de browser, maar hiernaast zijn er nog meer protocollen en services nodig:

- Interactive Connectivity Establishment (ICE):
  - Session Traversal Utilities for NAT (STUN)
  - Traversal Using Relays around NAT (TURN)

  Deze zijn nodig om een peer-to-peer connectie op te zetten en te onderhouden.
- Session Description Protocol (SDP) is een gegevensformaat dat wordt gebruikt om de parameters van de peer-to-peer verbinding te bepalen.
- Datagram Transport Layer Security (DTLS) wordt gebruikt om alle data transfers tussen de peers te beveiligen door middel van encryptie.
- Stream Control Transport Protocol (SCTP) en Secure Real-Time Transport Protocol (SRTP) worden gebruikt om Stream Multiplexing, congestion en flow control te kunnen voorzien.



Figuur 1: WebRTC protocol stack [10]

3.1.2 Een Peer-to-Peer Connectie maken

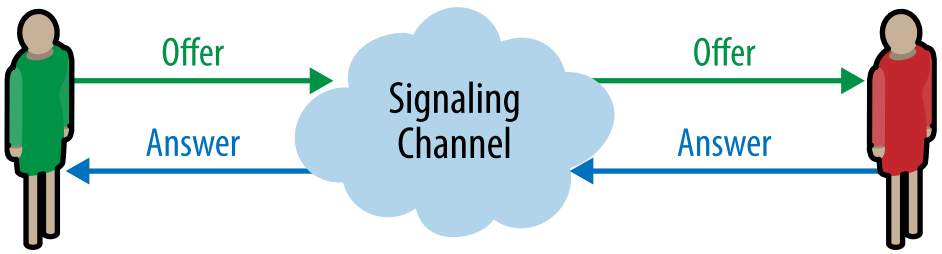
Meestal zitten twee peers niet in hetzelfde netwerk, waardoor ze niet direct bereikbaar zijn voor elkaar. Om een sessie te starten, moet het juiste IP-adres en de daarbij horende poort voor beide peers gevonden worden.

Wanneer een HTTP-verbinding geopend wordt naar de server, is er een impliciete veronderstelling dat de server luistert naar onze handshake. De server luistert dus altijd naar nieuwe verbindingen. Jammer genoeg is dit bij de connectie met een andere peer niet zo. De peer kan offline of onbereikbaar zijn. Daarnaast kan de peer druk bezig zijn en niet klaarstaan om een verbinding met een andere partij te maken.

Om een succesvolle peer-to-peer verbinding tot stand te brengen, moeten eerst enkele problemen opgelost worden:

- De andere peer moet op de hoogte zijn van de intentie om een peer-to-peer verbinding te openen, zodat de peer weet dat hij moet luisteren naar inkomende pakketten.
- Potentiële routeringstrajecten voor de peer-to-peer verbinding moeten geïdentificeerd zijn aan beide zijden van de verbinding.
- Informatie over de parameters van de streams en protocollen moeten uitgewisseld worden, alsook de gebruikte coderingen, enzovoort.

WebRTC lost gelukkig al één van deze problemen op. Zo voert het ingebouwde ICE-protocol de noodzakelijke routering en connectiviteitscontroles uit. Voordat de connectiviteitscontrole kan plaatsvinden, is het belangrijk om te weten of de andere peer bereikbaar is en of hij bereid is om de verbinding tot stand te brengen. De ene peer doet hiervoor een aanbod (offer) aan de andere peer en deze moet een antwoord teruggeven. Maar wat als de andere peer niet naar de binnenkomende pakketten luistert? Om de peer toch op de hoogte te brengen over de vraag om een verbinding te maken, is er op zijn minst een gedeeld signaleringskanaal nodig.



Figuur 2: Shared signaling channel [10]

WebRTC laat de keuze van het signaleringstransport en -protocol over aan de applicatie zelf. De standaard geeft opzettelijk geen aanbeveling of implementatie voor het signaleringskanaal, omdat het op deze manier mogelijk is om zonder problemen te kunnen samenwerken met verschillende signaleringsprotocollen die de bestaande communicatie-infrastructuur voeden, zoals de volgende:

- Session Initiation Protocol (SIP): Signaleringsprotocol op applicatieniveau dat vooral gebruikt wordt voor Voice over IP (VoIP) en videoconferenties over IP-netwerken.
- Jingle: Signaaluitbreiding voor het XMPP-protocol dat toegepast wordt voor sessiecontrole van VoIP en videoconferenties over IP-netwerken.
- ISDN User Part (ISUP): Signaleringsprotocol dat wordt gebruikt voor het tot stand brengen van telefoongesprekken in veel publiek geschakelde telefoonnetwerken over de hele wereld.

De applicatie kan ervoor kiezen om een van de bestaande signaleringsprotocollen en gateways te gebruiken, maar een andere mogelijkheid is om een eigen signaleringsdienst te implementeren met een eigen protocol.

Een signaleringsserver kan fungeren als gateway naar een bestaand communicatienetwerk, waarbij het dan de verantwoordelijkheid van het netwerk is om de doelpeer op de hoogte te brengen van een verbindingsaanbod en vervolgens het antwoord terug te routeren naar de WebRTC-client die de uitwisseling initieert.

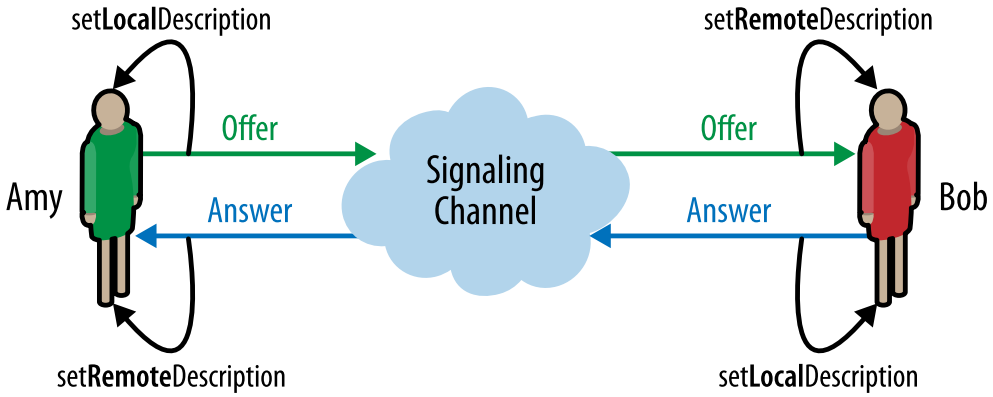
Daarnaast kan de applicatie ook zijn eigen aangepaste signaleringskanaal gebruiken, dat kan bestaan uit één of meer servers en een aangepast protocol om de berichten te communiceren. Als beide peers verbonden zijn met dezelfde signaleringsdienst, dan kan de dienst berichten tussen hen uitwisselen.

Elke WebRTC-applicatie heeft een signaleringsserver nodig om een verbinding tot stand te brengen. Er is ondertussen al een groeiende lijst van bestaande communicatie gateways die kunnen samenwerken met WebRTC. Dit kan bijvoorbeeld met Asterisk, een gratis en open-source framework dat zeer populair is. De applicatie kan ook makkelijk een aangepaste signalisatie-gateway ontwikkelen en implementeren als interoperabiliteit met andere netwerken niet vereist is. De keuze van signaleringstransport is zeer belangrijk omdat dit een significante invloed kan hebben op de reactietijd van het signaleringskanaal.

Session Description Protocol (SDP)

Om de parameters van de peer-to-peer verbinding te beschrijven, wordt het Session Description Protocol (SDP) gebruikt. Dit protocol levert zelf geen media, maar wordt gebruikt om het sessieprofiel te beschrijven. Deze beschrijving bevat een lijst van eigenschappen van de verbinding waaronder de soorten uit te wisselen media (audio, video, toepassingsgegevens), netwerktransporten, gebruikte codecs en hun instellingen, bandbreedte-informatie en andere metadata. De WebRTC-applicatie werkt niet rechtstreeks samen met SDP. Het JavaScript Session Establishment Protocol (JSEP) abstraheert de werking van SDP door een paar eenvoudige method calls in het RTCPeerConnection object.

Beide peers moeten een symmetrische workflow volgen om SDP-beschrijvingen van hun audio-, video- en andere gegevensstromen uit te wisselen.



Figuur 3: Offer/answer SDP exchange between peers [10]

Zoals te zien in figuur 3 gebeurt de sessiebeschrijving in verschillende stappen:

1. De initiator (Amy) registreert één of meerdere streams met haar lokale RTCPeerConnection object. Ze creëert een aanbod en stelt het in als haar lokale beschrijving van de sessie.
2. Amy stuurt vervolgens het gegenereerde sessieaanbod naar de andere peer (Bob).
3. Zodra Bob het aanbod ontvangen heeft, stelt hij Amy's beschrijving in als de remote beschrijving van de sessie. Hij registreert zijn eigen streams met zijn eigen RTCPeerConnection object, genereert een antwoord en stelt deze in als de lokale beschrijving van de sessie.
4. Bob stuurt het gegenereerde sessie antwoord terug naar Amy.
5. Als Amy het SDP-antwoord van Bob ontvangt, stelt zij zijn antwoord in als de remote beschrijving van haar oorspronkelijke sessie.

Interactive Connectivity Establishment (ICE)

De peers moeten in staat zijn om pakketten naar elkaar te routeren. Dit is in praktijk moeilijk vanwege de vele lagen firewalls en NAT-apparatuur tussen de peers.

Als beide peers zich in hetzelfde interne netwerk bevinden, is het makkelijker om een verbinding tot stand te brengen. Elke peer kan zijn IP-adres opvragen en deze samen met de juiste poort toevoegen aan de gegenereerde SDP-strings. De SDP-strings kunnen dan doorgestuurd worden naar de andere peer. Zodra de SDP-uitwisseling is voltooid, kunnen beide peers een directe peer-to-peer verbinding opzetten.

Hierboven staat al beschreven dat de peers in praktijk meestal niet in hetzelfde netwerk zitten. Wat gebeurt er dan? De voorgaande workflow herhalen is geen goede oplossing, aangezien de peer-to-peer verbindingen hierbij zouden falen. Een publiek routeringspad tussen de peers is nodig om de verbinding te kunnen maken.

Gelukkig beheert het WebRTC-framework het grootste deel van deze complexiteit. Elk RTCPeerConnection-object bevat een ICE-agent. Deze agent is verantwoordelijk voor zowel het verzamelen van lokale IP-adressen en poorten, als het uitvoeren van connectiviteitscontroles tussen de peers, als voor het versturen van keepalives voor de verbinding.

Zodra een sessiebeschrijving is ingesteld, begint de lokale ICE-agent automatisch alle mogelijke kandidaat IP-adressen en poorten voor de lokale peer te ontdekken. De ICE-agent vraagt de lokale IP-adressen op in het besturingssysteem. Indien geconfigureerd, vraagt de ICE-agent een externe STUN-server om het openbare IP-adres en de juiste poort van de peer op te halen. Indien geconfigureerd, voegt de ICE-agent de TURN-server toe als last resort kandidaat.

Telkens wanneer een nieuwe kandidaat (IP-adres en poort) wordt ontdekt, registreert de agent dit automatisch met het RTCPeerConnection-object en verwittigt de applicatie via een callback-functie.

Zodra de ICE-kandidaten door de andere peer zijn ontvangen, kan de tweede fase van het tot stand brengen van een peer-to-peer verbinding beginnen. Wanneer de sessiebeschrijving op het RTCPeerConnection-object is ingesteld, begint de ICE-agent connectiviteitscontroles te doen om te zien of de andere partij bereikbaar is.

De ICE-agent stuurt een STUN binding request, dat de andere peer moet bevestigen met een succesvol STUN-antwoord. Als dit lukt, is er een routeringstraject voor de peer-to-peer verbinding! Maar als alle kandidaten falen, dan wordt de RTCPeerConnection gemarkeerd als mislukt of valt de verbinding terug op een TURN relay server om de verbinding tot stand te brengen.

De connectiviteitscontroles worden in een bepaalde volgorde uitgevoerd. Eerst worden de lokale IP-adressen gecontroleerd, daarna de publieke en als laatste wordt de TURN-server gebruikt. Als de verbinding tot stand is gebracht, blijft de ICE-agent periodieke STUN-verzoeken uitgeven aan de andere peer. Dit dient als een keepalive van de verbinding.

Incremental Provisioning (Trickle ICE)

Het ophalen van lokale IP-adressen gaat snel, maar het opvragen van de STUN-server vereist een roundtrip naar de externe server, gevolgd door nog een ronde van STUN-connectiviteitscontroles tussen de individuele peers. Trickle ICE is een uitbreiding van het ICE-protocol dat incrementele vergaring en connectiviteitscontroles tussen de peers mogelijk maakt.

Het kernidee is zeer eenvoudig:

- Beide peers wisselen SDP-aanbiedingen uit zonder ICE-kandidaten.
- ICE-kandidaten worden via het signaleringskanaal verzonden wanneer ze worden ontdekt.
- ICE-connectiviteitscontroles worden uitgevoerd zodra de nieuwe kandidaat beschrijving beschikbaar is.

Kortom, wordt er vertrouwd op het signaleringskanaal om incrementele updates aan de andere peer te leveren, wat het proces helpt te versnellen. Trickle ICE genereert meer verkeer over het signaleringskanaal, maar het kan een aanzienlijke verbetering opleveren in de tijd die nodig is om de peer-to-peer verbinding op te zetten. Daarom is het ook de aanbevolen strategie voor alle WebRTC-applicaties.

ICE connectiviteitsstatus

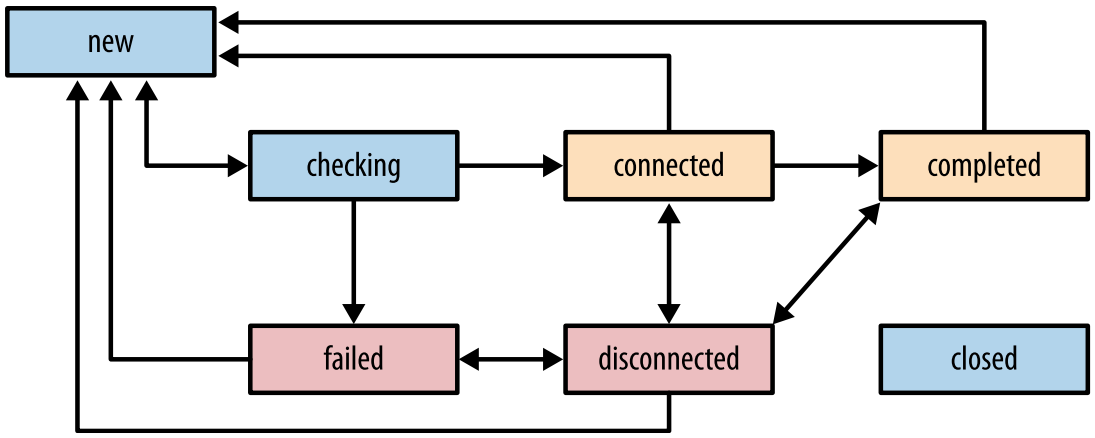
Niet elke verbinding zal slagen en het is belangrijk dat het probleem geïsoleerd en opgelost wordt. Daarom is het belangrijk om de status van de ICE-agent op te vragen en hierover meldingen te ontvangen.

De status van het proces omtrent het verzamelen van kandidaten kan in 3 verschillende toestanden verkeren:

- New: Het object is zojuist aangemaakt en er heeft zich nog geen netwerkvorming voorgedaan.
- Gathering: De ICE-agent is bezig met het verzamelen van lokale kandidaten.
- Complete: De ICE-agent heeft het verzamelproces voltooid.

Ook de status van de peer-to-peer verbinding kan zich in verschillende toestanden bevinden:

- New: De ICE-agent is kandidaten aan het verzamelen en/of wacht tot kandidaten worden geleverd.
- Checking: De ICE-agent heeft kandidaten ontvangen en is kandidaat-paren aan het controleren, maar het heeft nog geen verbinding gevonden. Daarnaast kan hij ook nog bezig zijn met het verzamelen van kandidaten.
- Connected: Er is een bruikbare verbinding gevonden voor alle componenten, maar er worden nog altijd andere kandidaat-paren gecontroleerd om te zien of er een betere verbinding is. De ICE-agent kan nog steeds kandidaten verzamelen.
- Completed: De ICE-agent is klaar met verzamelen en controleren en heeft een verbinding gevonden voor alle componenten.
- Failed: Er zijn geen kandidaat-paren gevonden waarmee een verbinding kan gemaakt worden voor alle componenten.
- Disconnected: De aanwezigheidscontroles zijn mislukt voor één of meer componenten. Dit is agressiever dan failed en kan met tussenpozen optreden (en zichzelf oplossen zonder actie te ondernemen) op een onbetrouwbaar netwerk.
- Closed: De ICE-agent heeft zich afgesloten en reageert niet langer op STUN-verzoeken.



Figuur 4: ICE agent connectivity states and transitions [10]

Een WebRTC-sessie kan meerdere streams nodig hebben om audio, video en applicatiegegevens te leveren. Een succesvolle verbinding moet dus in staat zijn om connectiviteit tot stand te brengen voor alle gevraagde streams. Door de onbetrouwbare connectiviteit van peer-to-peer connecties, zijn er geen garanties dat eens de verbinding tot stand is gebracht, deze ook zo blijft. De verbinding kan periodiek heen en weer schakelen tussen verbonden en verbroken toestand, terwijl de ICE-agent het best mogelijke pad probeert te vinden om de connectiviteit opnieuw tot stand te brengen.

Het primaire doel voor de ICE-agent is het identificeren van een levensvatbaar routeringspad tussen de peers. Maar daar blijft het niet bij. Zelfs wanneer er een verbinding gemaakt is, kan de ICE-agent periodiek andere kandidaten proberen, om te zien of hij betere prestaties kan leveren via een alternatieve route.

3.1.3 Media en Applicatie Data leveren

Op dit moment hebben beide peers een onbewerkte UDP-verbinding met elkaar openstaan, die een datagram transport biedt, maar dat is niet voldoende. Zonder flow control, congestion control, foutcontrole en een mechanisme voor bandbreedte en reactietijd schatting, kunnen we gemakkelijk het netwerk overweldigen, wat zou leiden tot verslechterde prestaties.

Bovendien verstuurt UDP gegevens ongecodeerd, terwijl WebRTC vereist dat we alle communicatie versleutelen. Om dit op te vangen, legt WebRTC verschillende extra protocollen bovenop UDP:

- Datagram Transport Layer Security (DTLS) is belangrijk voor het coderen van mediagegevens en voor het veilig transporteren van applicatiegegevens.
- Secure Real-Time Transport Protocol (SRTP) wordt gebruikt om audio- en videostromen te transporteren.
- Stream Control Transport Protocol (SCTP) wordt gebruikt om applicatiegegevens te transporteren.

Veilige communicatie met DTLS

WebRTC vereist dat alle overgedragen gegevens tijdens de overdracht worden versleuteld. Het Transport Layer Security (TLS) protocol zou hiervoor perfect zijn, maar dit protocol kan niet over UDP gebruikt worden. In plaats daarvan gebruikt WebRTC DTLS, dat gelijkwaardige veiligheidsgaranties biedt.

DTLS is ontworpen om zo veel mogelijk op TLS te lijken. In feite is DTLS TLS, maar met een minimaal aantal aanpassingen om het compatibel te maken met het datagram transport dat door UDP wordt aangeboden. In het bijzonder pakt DTLS de volgende problemen aan:

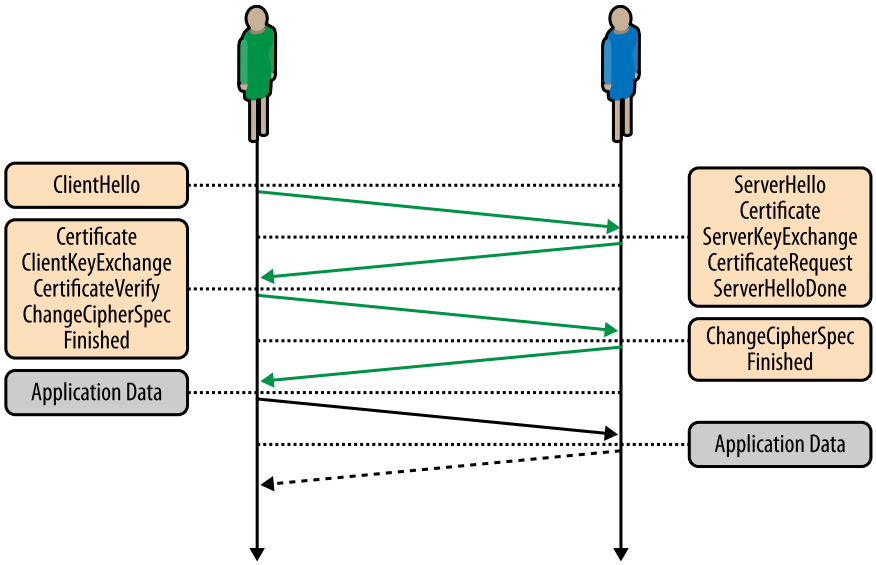
- TLS vereist betrouwbare, in volgorde en fragmentatie vriendelijke levering van de handshake records.
- TLS-integriteit controles kunnen mislukken als records over meerdere pakketten worden gefragmenteerd.
- TLS-integriteit controles kunnen mislukken als records niet in de juiste volgorde worden verwerkt.

Er is geen eenvoudige alternatieve oplossing om de TLS-handshake sequentie te herstellen. Daarom breidt DTLS het basis TLS-protocol uit door een expliciete fragment-offset en een sequentienummer toe te voegen voor elk handshake record. Hierdoor wordt voldaan aan de eis van levering in volgorde en kunnen grote records over pakketten worden gefragmenteerd en door de andere peer opnieuw worden samengevoegd.

DTLS-handshake records worden verzonden in de exacte volgorde die door het TLS-protocol wordt gespecifieerd.

Tenslotte moet DTLS ook omgaan met pakketverlies. Beide zijden gebruiken eenvoudige timers om handshake records opnieuw uit te zenden als het antwoord niet binnen een verwacht interval wordt ontvangen.

De combinatie van het record volgordenummer, de offset en de timer stelt DTLS in staat de handshake over UDP uit te voeren. Om deze reeks te voltooien, genereren beide netwerkpeers zelfondertekende certificaten en volgen dan het normale TLS-handshake protocol.



Figuur 5: Peer-to-peer handshake over DTLS [10]

De DTLS-handshake vereist twee roundtrips om te voltooien. Dit is een belangrijk aspect om in gedachten te houden, omdat het extra reactietijd toevoegt aan het opzetten van de peer-to-peer verbinding.

De WebRTC-client genereert automatisch zelf ondertekende certificaten voor elke peer. Als gevolg daarvan is er geen certificaatketen om te verifiëren. DTLS biedt versleuteling en integriteit, maar de authenticatie gebeurt door de applicatie. DTLS voegt twee belangrijke regels toe om rekening te houden met mogelijke fragmentatie en verwerking van gewone records in een andere volgorde:

- DTLS-records moeten in een enkel netwerkpakket passen.
- Een block cipher moet gebruikt worden voor het coderen van recordgegevens.

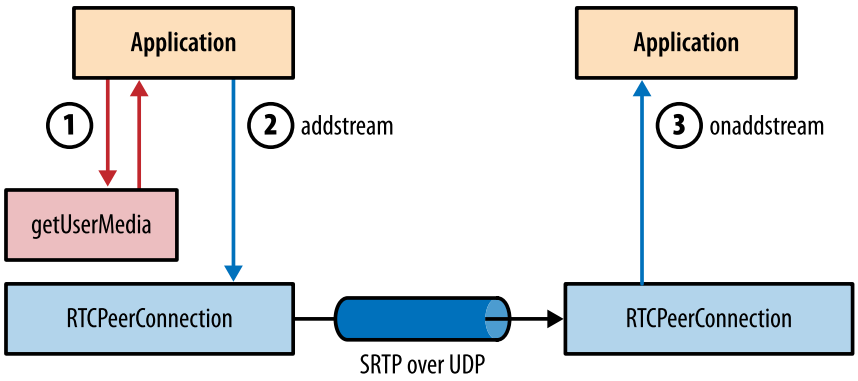
De DTLS-handshake die tussen twee WebRTC-clients wordt uitgevoerd is gebaseerd op zelfondertekende certificaten. Als gevolg daarvan kunnen de certificaten zelf niet gebruikt worden om de peer te authenticeren, aangezien er geen expliciete vertrouwensketen is om te verifiëren.

Indien nodig moet de WebRTC-applicatie haar eigen authenticatie en identiteitsverificatie van de deelnemende peers uitvoeren:

- Een webapplicatie kan zijn bestaande systeem voor identiteitsverificatie gebruiken. Deze verificatie gebeurt dan voor het opzetten van de WebRTC-sessie.
- Elke deelnemende peer kan zijn identiteitsprovider specificeren bij het genereren van de SDP-aanbieding/het SDP-antwoord. Wanneer het SDP-bericht wordt ontvangen, kan de tegengestelde peer contact opnemen met de gespecificeerde identiteitsprovider om het ontvangen certificaat te verifiëren. Het identiteitsprovider mechanisme wordt nog steeds actief besproken en ontwikkeld in de W3C WebRTC-workgroup.

Media afleveren met SRTP en SRTCP

De WebRTC-applicatie specifieert de mediabeperkingen om de streams te verwerven en registreert ze dan met het RTCPeerConnection-object. Van daar wordt de rest afgehandeld door de WebRTC media- en netwerk-engines die door de browser worden geleverd.



Figuur 6: Audio and video delivery via SRTP over UDP [10]

De applicatie heeft geen directe controle over hoe een video geoptimaliseerd of afgeleverd wordt aan de andere peer. Deze ontwerpkeuze is opzettelijk. Het afleveren van een hoge kwaliteit, real-time audio en video stream over een onbetrouwbaar transport met fluctuerende bandbreedte en pakket reactietijd is een probleem. De browser lost dit voor ons op:

- Ongeacht de kwaliteit en grootte van de geleverde media streams, implementeert de netwerk stack zijn eigen flow en congestion control algoritmes. Elke verbinding begint met het streamen van audio en video met een lage bitrate en begint dan de kwaliteit van de streams aan te passen aan de beschikbare bandbreedte.
- De media- en netwerk-engines passen de kwaliteit van de streams dynamisch aan gedurende de levensduur van de verbinding.

Een audio- of videostream kan worden geleverd met een lagere kwaliteit dan die van de oorspronkelijke stream die door de toepassing is verkregen. Het omgekeerde is echter niet waar: WebRTC zal de kwaliteit van de stream niet opwaarderen.

WebRTC is niet de eerste toepassing die tegen de uitdaging aanloopt om audio- en videoverzending in real-time over IP-netwerken te implementeren. Daarom maakt WebRTC opnieuw gebruik van bestaande transportprotocollen die worden gebruikt door VoIP-telefoons, communicatie gateways en talrijke commerciële en open-source communicatiediensten:

- Secure Real-Time Transport Protocol (SRTP) beveiligt het gestandaardiseerde formaat voor de levering van real-time gegevens over IP-netwerken.
- Secure Real-Time Control Transport Protocol (SRTCP) beveiligt het controleprotocol voor de levering van verzender- en ontvangerstatistieken en controle-informatie voor een SRTP-stream.

Real-Time Transport Protocol (RTP) wordt gedefinieerd door RFC-3550. WebRTC vereist echter dat alle communicatie onderweg wordt versleuteld, daarom wordt het veilige profiel (RFC-3711) van RTP gebruikt.

SRTP definieert een standaard pakketformaat voor het leveren van audio en video over IP-netwerken. Op zichzelf biedt SRTP geen mechanisme of garanties voor tijdigheid, betrouwbaarheid of foutherstel van de overgedragen gegevens. In plaats daarvan omhult het eenvoudig de audio- en videoframes met extra metadata om de ontvanger te helpen bij de verwerking van elke stream.

Het SRTP-pakket bevat alle essentiële informatie die de media engine nodig heeft om de stream af te spelen. De verantwoordelijkheid om te controleren hoe de individuele SRTP-pakketten worden afgeleverd valt echter toe aan het SRTCP-protocol, dat een afzonderlijk, out-of-band feedback kanaal implementeert voor elke media stream.

SRTCP houdt het aantal verzonden en verloren bytes en pakketten bij, het laatst ontvangen sequentienummer en andere SRTP-statistieken. Beide peers wisselen deze gegevens periodiek uit en gebruiken ze om de verzendsnelheid, coderingskwaliteit en andere parameters van elke stream aan te passen.

SRTP en SRTCP lopen dus rechtstreeks over UDP en werken samen om de real-time levering van audio- en videostromen, die door de applicatie worden geleverd, aan te passen en te optimaliseren.

Applicatiegegevens afleveren met SCTP

Voor de overdracht van applicatiedata wordt het Stream Control Transmission Protocol (SCTP) gebruikt. Dit protocol werkt bovenop de DTLS tunnel die eerder werd opgezet.

Wat zijn nu de WebRTC vereisten voor het protocol?

- Multiplexing van meerdere onafhankelijke kanalen moet worden ondersteund.
  - Elk kanaal moet in-order of out-of-order levering ondersteunen.
  - Het kanaal moet betrouwbare of onbetrouwbare aflevering ondersteunen.
- Er moet een bericht-georiënteerde API beschikbaar zijn.
- Het moet flow- en congestion controlemechanismen implementeren.
- Vertrouwelijkheid en integriteit van de overgedragen gegevens moeten gewaarborgd zijn.

Het laatste criterium wordt reeds voldaan door het gebruik van DTLS. Alle applicatiegegevens worden geëncrypteerd binnen de payload van het record en de vertrouwelijkheid en integriteit zijn gewaarborgd. Aan de resterende eisen is echter niet altijd te voldoen.

	TCP	**UDP**	**SCTP**
Betrouwbaarheid	Betrouwbaar	Onbetrouwbaar	Configureerbaar
Afleveringsmanier	Geordend	Ongeordend	Configureerbaar
Transmissie	Byte-georiënteerd	Bericht-georiënteerd	Bericht-georiënteerd
Flow control	Ja	Nee	Ja
Congestion control	Ja	Nee	Ja

Tabel 1: Het verschil tussen TCP, UDP en SCTP [10]

SCTP is een transportprotocol, vergelijkbaar met TCP en UDP, dat rechtstreeks bovenop het IP-protocol kan draaien. In het geval van WebRTC wordt SCTP echter getunneld over een beveiligde DTLS-tunnel, die zelf bovenop UDP loopt.
SCTP gebruikt bepaalde concepten en terminologieën:

- Association: Een synoniem voor een verbinding.
- Stream: Een eenrichtingskanaal waarbinnen applicatieberichten in volgorde worden afgeleverd, tenzij het kanaal geconfigureerd is om de niet-geordende afleverservice te gebruiken.
- Message: Applicatiegegevens die aan het protocol worden voorgelegd.
- Chunk: De kleinste eenheid van communicatie binnen een SCTP-pakket.

Een enkele SCTP-associatie tussen twee eindpunten kan meerdere onafhankelijke streams dragen, die elk communiceren door applicatieberichten over te brengen. Elk bericht kan op zijn beurt worden opgesplitst in één of meer chunks, die worden afgeleverd binnen SCTP-pakketten en dan aan het andere eind weer worden samengevoegd. Deze beschrijving kan bekend klinken, aangezien de concepten identiek zijn aan die van de HTTP/2 framing laag.

Een SCTP-pakket bestaat uit een gemeenschappelijke header en één of meer controle- of gegevenschunks. De header bevat 12 bytes aan gegevens, die de bron- en bestemmingspoort identificeren, een willekeurig gegenereerde verificatietag voor de huidige SCTP-associatie en de checksum voor het hele pakket.

Bit	+0..7	+8..15	+16..23	+24..31
0	Source Port		Destination Port	
32	Verification Tag			
64	Checksum			
96	Type (0)	Reserved	U B E	Length
128	Transmission sequence number (TSN)			
160	Stream identifier		Stream sequence number	
192	Payload protocol identifier (PPID)			
224	*Payload*			

Figuur 7: SCTP header and data chunk [10]

In de figuur hierboven wordt een SCTP-pakket met een enkele gegevenschunk getoond:

- Alle data chunks hebben een 0x0 datatype.
- De unordered (U) bit geeft aan of dit een ongeordende data chunk is.
- B en E bits worden gebruikt om het begin en einde aan te geven van een bericht dat verdeeld is over meerdere chunks:
    - B=1, E=0 geeft het eerste fragment van een bericht aan.
    - B=0, E=0 wordt gebruikt om het middenstuk aan te duiden.
    - B=0, E=1 geeft het laatste fragment aan.
    - B=1, E=1 duidt een niet-gefragmenteerd bericht aan.
- De lengte geeft de grootte van de data chunk weer, die de header omvat. De header bestaat uit 16 bytes en hier wordt de grootte van de payload nog bij opgeteld.
- Transmission sequence number (TSN) is een 32-bit nummer dat intern door SCTP wordt gebruikt om de ontvangst van het pakket te bevestigen en zo dubbele leveringen detecteert.
- De stream identifier geeft de stream aan waartoe de chunk behoort.
- Het stream sequence number is een berichtnummer dat automatisch verhoogd voor de bijbehorende stream. Gefragmenteerde berichten dragen hetzelfde sequentienummer.
- Payload protocol identifier (PPID) is een aangepast veld dat door de applicatie wordt ingevuld om aanvullende metadata over de overgedragen chunk mee te delen.

Elke SCTP-verbinding vereist een handshake-sequentie die vergelijkbaar is met die van TCP. SCTP implementeert ook TCP-vriendelijke mechanismen voor flow en congestion control. Beide protocollen gebruiken dezelfde initiële congestion window grootte en implementeren vergelijkbare logica om dit te laten groeien en verkleinen zodra de verbinding in de congestion-vermijdende fase komt.

Ondertussen zijn er al heel wat WebRTC vereisten voldaan, maar zelfs met al deze functionaliteit zijn er nog steeds een paar vereiste eigenschappen tekort:

- De basis SCTP-standaard (RFC-4960) voorziet een mechanisme voor ongeordende aflevering van berichten, maar geen faciliteiten voor het configureren van de betrouwbaarheid van elk bericht. Om dit te verhelpen moeten WebRTC-clients ook de Partial Reliability Extension (RFC-3758) gebruiken, die het SCTP-protocol uitbreidt en de verzender in staat stelt aangepaste afleveringsgaranties te implementeren.
- SCTP biedt geen faciliteiten voor het prioriteren van individuele streams. Er zijn geen velden in het protocol om de prioriteit aan te geven. Als gevolg daarvan moet deze functionaliteit hoger in de stack worden geïmplementeerd.

3.1.4 DataChannel

DataChannel maakt de uitwisseling van applicatiedata tussen peers mogelijk. Zodra de RTCPeerConnection tot stand is gebracht, kunnen verbonden peers één of meer kanalen openen om tekst of binaire gegevens uit te wisselen.

De DataChannel API komt overeen met die van een WebSocket. Elk opgezet kanaal vuurt dezelfde onerror, onclose, onopen en onmessage callbacks af en stelt dezelfde binaryType, bufferedAmount en protocolvelden beschikbaar op het kanaal.

Aangezien DataChannel peer-to-peer is en over een flexibeler transport protocol loopt, biedt het ook een aantal extra mogelijkheden die WebSocket niet heeft. Enkele belangrijke verschillen worden hieronder opgenoemd:

- In tegenstelling tot de WebSocket-constructor, die de URL van de WebSocket-server verwacht, is DataChannel een factory method op het RTCPeerConnection-object.
- Elke peer kan een nieuwe DataChannel-sessie starten. De ondatachannel callback wordt afgevuurd wanneer een nieuwe DataChannel-sessie start.
- Elke DataChannel kan met aangepaste afleverings- en betrouwbaarheid semantiek geconfigureerd worden.

DataChannel versus WebSocket API's

De DataChannel API is een superset van de WebSocket API. Hierdoor zijn de WebSocket callbacks, flags, optimalisaties voor verwerking van tekst en binaire data direct van toepassing op de DataChannel API.

	WebSocket	**DataChannel**
Encryptie	Configureerbaar	Altijd
Betrouwbaarheid	Betrouwbaar	Configureerbaar
Afleveringsmanier	Geordend	Configureerbaar
Multiplexed	Nee (extensie)	Ja
Transmissie	Bericht-georiënteerd	Bericht-georiënteerd
Binary transfers	Ja	Ja
UTF-8 transfers	Ja	Ja
Compressie	Nee (extensie)	Nee

Tabel 2: WebSocket vs. DataChannel [10]

Het grootste verschil tussen WebSocket en DataChannel is natuurlijk het onderliggende transport. WebSocket draait bovenop TCP, dat zorgt voor een betrouwbare aflevering van elk bericht in de juiste volgorde terwijl DataChannel bovenop drie protocollen ligt:

- UDP zorgt voor peer-to-peer connectiviteit.
- DTLS zorgt voor encryptie van overgedragen gegevens.
- SCTP zorgt voor multiplexing, flow en congestion control en andere functies.

DataChannel kan worden geconfigureerd om dezelfde betrouwbaarheid en in-order berichtgaranties te leveren als WebSocket. Hoewel de echte kracht van DataChannel net ligt in het feit dat het niet de in-order en betrouwbare aflevering semantiek hoeft te volgen. Elk kanaal kan zijn eigen leverings- en betrouwbaarheidsvereisten specificeren en de gegevens kunnen direct peer-to-peer worden overgedragen.

Setup

Ongeacht het type overgedragen gegevens moeten de twee deelnemende peers de volledige “offer/answer”-workflow doorlopen, onderhandelen over de gebruikte protocollen en poorten en hun connectiviteitscontroles met succes voltooien.

Zoals eerder vermeld, verlopen mediatransfers over SRTP, terwijl DataChannel het SCTP-protocol gebruikt. Als gevolg hiervan, moeten er bepaalde parameters gespecifieerd worden bij de verbinding. Wanneer de initiërende peer voor het eerst het verbindingsaanbod doet, of wanneer het antwoord door de andere peer wordt gegenereerd, moeten de twee de parameters voor de SCTP-associatie specifiëren binnen de gegenereerde SDP-strings.

Zoals voorheen handelt het RTCPeerConnection-object de noodzakelijke generatie van de SDP-parameters af, zolang een van de peers een DataChannel registreert voordat de SDP-beschrijving van de sessie wordt gegenereerd. In feite kan de toepassing een peer-to-peer verbinding tot stand brengen die alleen gegevens bevat door expliciete beperkingen in te stellen om audio- en video-overdracht uit te schakelen.

Het SDP-fragment zegt niets over de parameters van elke DataChannel, bv. protocol, betrouwbaarheid, in-order of out-of-order flags. Als gevolg hiervan stuurt de WebRTC-client die de verbinding initieert ook een DATA_CHANNEL_OPEN-bericht dat het type, de betrouwbaarheid, het gebruikte applicatieprotocol en andere parameters van het kanaal beschrijft, voordat applicatiegegevens kunnen worden verzonden.

Zodra de kanaalparameters zijn doorgegeven, kunnen beide peers beginnen met het uitwisselen van applicatiegegevens. Elk tot stand gebracht kanaal wordt afgeleverd als een onafhankelijke SCTP-stream. De kanalen worden gemultiplexed over dezelfde SCTP-associatie, waardoor head-of-line-blokkering tussen de verschillende streams wordt vermeden en gelijktijdige aflevering van meerdere kanalen over dezelfde SCTP-associatie mogelijk wordt gemaakt.

Berichtvolgorde en betrouwbaarheid

DataChannel maakt peer-to-peer overdracht van willekeurige applicatiegegevens mogelijk via een WebSocket-compatibele API. Dit is op zichzelf al een unieke en krachtige eigenschap. DataChannel biedt echter ook een veel flexibeler transport, dat ons in staat stelt om de aflever semantiek van elk kanaal aan te passen aan de vereisten van de applicatie en het type gegevens dat wordt overgedragen:

- Berichten kunnen in volgorde afgeleverd worden, maar ook willekeurig.
- Het kan een betrouwbare of gedeeltelijk betrouwbare aflevering van berichten bieden.

Het configureren van het kanaal om in-order en betrouwbare aflevering te gebruiken is gelijk aan TCP: dezelfde afleveringsgaranties als een gewone WebSocket-verbinding. DataChannel biedt echter ook twee verschillende policies voor het configureren van gedeeltelijke betrouwbaarheid van elk kanaal:

- Partially reliable delivery with retransmit: Berichten worden niet vaker opnieuw verzonden dan door de applicatie is opgegeven.
- Partially delivery with timeout: Berichten worden niet opnieuw verzonden na een door de applicatie opgegeven tijdsduur (in milliseconden).

Beide strategieën worden geïmplementeerd door de WebRTC-client, wat betekent dat de applicatie alleen maar hoeft te beslissen welk leveringsmodel ze willen gebruiken. Daarbij moeten de juiste parameters op het kanaal ingesteld worden. Het is niet nodig om applicatietimers te beheren.

Op de figuur hieronder zijn de verschillende configuratieopties te zien:

	Ordered	Reliable	Partial reliability policy
Ordered + reliable	yes	yes	n/a
Unordered + reliable	no	yes	n/a
Ordered + partially reliable (retransmit)	yes	partial	retransmission count
Unordered + partially reliable (retransmit)	no	partial	retransmission count
Ordered + partially reliable (timed)	yes	partial	timeout (ms)
Unordered + partially reliable (timed)	no	partial	timeout (ms)

Figuur 8: DataChannel reliability and delivery configurations [10]

Elk DataChannel kan dus worden geconfigureerd met aangepaste volgorde- en betrouwbaarheidsparameters en de peers kunnen meerdere kanalen openen, die allen worden gemultiplexed over dezelfde SCTP-associatie. Bijgevolg is elk kanaal onafhankelijk van de andere kanalen en kunnen de peers verschillende kanalen gebruiken voor verschillende soorten gegevens.

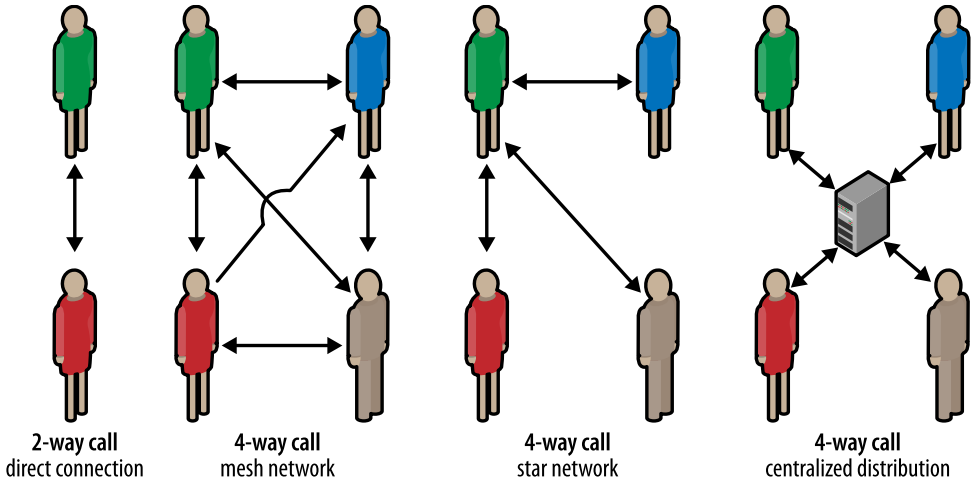
3.1.5 WebRTC Use Cases en Performance

Het implementeren van een peer-to-peer transport met een lage reactietijd is een niet alledaagse technische uitdaging. Er zijn NAT traversals en connectiviteitscontroles, signalering, beveiliging en talloze andere details om te regelen. WebRTC regelt al het bovenstaande en is daarom misschien wel een van de belangrijkste toevoegingen aan het webplatform sinds het ontstaan hiervan. In feite gaat het niet alleen om de afzonderlijke onderdelen van WebRTC, maar om het feit dat alle componenten samenwerken om een eenvoudige en uniforme API te leveren voor het bouwen van peer-to-peer toepassingen in de browser.

Maar zelfs met alle ingebouwde diensten vereist het ontwerpen van efficiënte en performante peer-to-peer toepassingen nog steeds een grote hoeveelheid denkwerk en planning, omdat peer-to-peer op zichzelf niet bekend staat voor zijn hoge prestaties. De grotere variabiliteit in bandbreedte en reactietijd tussen de peers, de hoge eisen die aan de overdracht van media worden gesteld en de eigenaardigheden van onbetrouwbare levering maken het een nog moeilijkere technische uitdaging.

Een architectuur voor meerdere partijen

Een enkele peer-to-peer verbinding kan gemakkelijk een aanzienlijk deel van de bandbreedte in beslag nemen. Daarom moeten toepassingen voor meerdere partijen goed nadenken over de architectuur van hoe de individuele streams worden samengevoegd en verdeeld tussen de peers.



Figuur 9: Distribution architecture for an N-way call [10]

Eén-op-één verbindingen zijn gemakkelijk te beheren en implementeren. De peers praten rechtstreeks met elkaar en er is geen verdere optimalisatie nodig. Als dezelfde strategie echter wordt uitgebreid tot een N-weg gesprek, waarbij elke peer verantwoordelijk is voor de verbinding met elke andere partij (een mesh-netwerk), zou dat resulteren in verbindingen voor elke peer. Als bandbreedte schaars is, zoals vaak het geval is als gevolg van de lagere uplink-snelheden, zal dit type architectuur de verbindingen van de meeste gebruikers snel verzadigen met slechts een paar deelnemers.

Een alternatieve strategie is om een stertopologie te gebruiken, waarbij de individuele peers verbinding maken met een supernode, die dan verantwoordelijk is voor het distribueren van de streams naar alle aangesloten partijen. Op deze manier hoeft slechts één peer de kosten te betalen voor het afhandelen en distribueren van streams en alle anderen praten rechtstreeks met de supernode.

Een supernode kan een andere peer zijn of het kan een speciale dienst zijn die geoptimaliseerd is voor het verwerken en distribueren van real-time data. Welke strategie het meest geschikt is, hangt af van de context en de applicatie. In het eenvoudigste geval, kan de initiator optreden als supernode. Een betere strategie zou kunnen zijn om de peer met de beste beschikbare verwerkingscapaciteit te kiezen.

Tenslotte zou de supernode een specifieke dienst kunnen zijn en zelfs een dienst van derden. WebRTC maakt peer-to-peer communicatie mogelijk, maar dit betekent niet dat er geen plaats is voor gecentraliseerde infrastructuur. Individuele peers kunnen peer-verbindingen tot stand brengen met een proxyserver en nog steeds profiteren van zowel de WebRTC-transportinfrastructuur als de extra diensten die door de server worden aangeboden.

Infrastructuur en capaciteitsplanning

Naast het plannen en anticiperen op de bandbreedte-eisen van individuele peer-verbindingen, zal elke WebRTC-toepassing een gecentraliseerde infrastructuur nodig hebben voor signalering, NAT en firewall traversal, identiteitsverificatie en andere aanvullende diensten die door de toepassing worden aangeboden.

Alle signalering wordt gedaan vanuit de applicatie, wat betekent dat de applicatie ten minste de mogelijkheid moet bieden om berichten naar de andere peer te verzenden en te ontvangen. Het volume van de verzonden signaleringsgegevens zal variëren naargelang van het aantal gebruikers, het protocol, de codering van de gegevens en de frequentie van de updates. Ook de reactietijd van de signaleringsdienst heeft een grote invloed op de “call setup”-tijd en andere signaleringsuitwisselingen.

- Gebruik daarom een transport met lage reactietijd, zoals WebSocket of SSE met XHR.
- Schat en voorzie voldoende capaciteit om de noodzakelijke signalerings-snelheid voor alle gebruikers van uw toepassing aan te kunnen.

De meeste WebRTC toepassingen zullen een STUN-server nodig hebben om de nodige IP-lookups uit te voeren om de peer-to-peer verbinding tot stand te brengen. Het goede nieuws is dat de STUN-server alleen wordt gebruikt voor het opzetten van de verbinding, maar desalniettemin moet hij het STUN-protocol aanspreken en moet het de nodige query load aankunnen.

Zelfs met STUN in werking zal 8% tot 10% van de peer-to-peer verbindingen waarschijnlijk falen. Dit komt door eigenaardigheden in hun netwerkbeleid. Zo zou een netwerkbeheerder UDP bijvoorbeeld kunnen blokkeren voor alle gebruikers op het netwerk.

Een applicatie voor meerdere partijen kan een gecentraliseerde infrastructuur nodig hebben om de levering van vele streams te optimaliseren en aanvullende diensten te leveren als onderdeel van de RTC-ervaring. In sommige opzichten hebben multiparty gateways dezelfde rol als TURN, maar er zijn ook verschillen. In tegenstelling tot TURN-servers, die fungeren als eenvoudige packet proxies, kan een slimme proxy aanzienlijk meer CPU en GPU middelen vereisen om elke individuele stream te verwerken voordat de uiteindelijke output wordt doorgestuurd naar elke aangesloten partij.

Data efficiëntie

De bitrate van de mediastromen worden door WebRTC audio- en video-engines dynamisch aangepast aan de omstandigheden van de netwerkverbinding tussen de peers. De applicatie kan de mediabeperkingen instellen en bijwerken (bv. videoresolutie en framerate) en de engines doen de rest.

Helaas kan dit niet gezegd worden van DataChannel, dat is ontworpen om willekeurige applicatiegegevens te transporteren. Net als WebSocket accepteert de DataChannel API binaire en UTF-8 gecodeerde applicatiedata, maar het past geen verdere bewerking toe om de grootte van de overgedragen data te verkleinen. Het is de verantwoordelijkheid van de WebRTC-applicatie om de binaire payloads te optimaliseren en de UTF-8 inhoud te comprimeren.

Verder moeten WebRTC-applicaties rekening houden met de extra overhead van de UDP-, DTLS- en SCTP-protocollen.

3.1.6 Performance Checklist

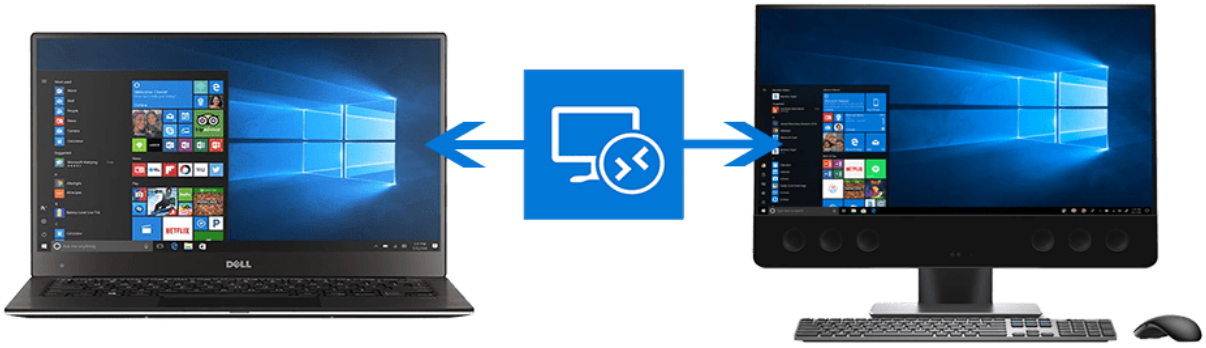
Peer-to-peer architecturen brengen hun eigen unieke set van prestatie-uitdagingen voor de toepassing met zich mee. Directe één-op-één communicatie is relatief eenvoudig, maar de zaken worden veel ingewikkelder wanneer er meer dan twee partijen bij betrokken zijn. Dit is vooral voelbaar in de prestaties. Daarom is het belangrijk om rekening te houden met de volgende criteria:

- Signaling service:
    - Gebruik een transport met lage reactietijd.
    - Zorg voor voldoende capaciteit.
    - Overweeg het gebruik van signalering over DataChannel zodra de verbinding tot stand is gebracht.
- Firewall en NAT traversal:
    - Zorg voor een STUN-server bij het initiëren van RTCPeerConnection.
    - Gebruik waar mogelijk Trickle ICE (meer signalering, maar snellere setup).
    - Zorg voor een TURN-server voor het doorverbinden van mislukte peer-to-peer verbindingen.
    - Anticipeer en voorzie voldoende capaciteit voor TURN-relays.
- Verspreiding van de gegevens:
    - Overweeg het gebruik van een supernode voor communicatie tussen meerdere partijen.
    - Optimaliseer de ontvangen data bij de supernode alvorens het door te sturen naar de andere peers.
- Data efficiëntie:
    - Specifieer geschikte mediabeperkingen voor spraak- en videostreams.
    - Optimaliseer binaire payloads die over DataChannel worden verzonden.
    - Comprimeer UTF-8 inhoud die over DataChannel wordt verzonden.
    - Monitor de hoeveelheid gebufferde data op het DataChannel en doe aanpassingen bij veranderingen in de toestand van de netwerk link.

- Levering en betrouwbaarheid:
    - Gebruik levering out-of-order om head-of-line blokkering te vermijden.
    - Als levering in volgorde wordt gebruikt, minimaliseer dan de berichtgrootte om het effect van head-of-line blokkering te verminderen.
    - Stuur kleine berichten om de impact van pakketverlies op gefragmenteerde applicatieberichten te minimaliseren.
    - Stel het juiste aantal heruitzendingen en time-outs in voor gedeeltelijk betrouwbare aflevering. De juiste instellingen hangen af van de berichtgrootte, het type applicatiegegevens en de reactietijd tussen de peers.

3.2 Remote access control

Remote access control is de mogelijkheid om een computer die zich niet in uw fysieke aanwezigheid bevindt, te beheren. Hierdoor kan er dus op de computer gewerkt worden vanop afstand.



Figuur 10: Remote access [11]

In het praktische deel van deze bachelorproef wordt er gekeken hoe de muis kan beheerd worden en hoe er geklikt kan worden vanop afstand.

3.2.1 Wat kan er misgaan?

Werken op afstand brengt natuurlijk extra risico's met zich mee. Omdat het moeilijk is om te zien wat er op een endpoint gebeurt, is het ook niet evident om op bedreigingen te reageren. Daarnaast is de verbinding tussen het apparaat op afstand en het bedrijfsnetwerk kwetsbaarder voor bedreigingen. Dit komt omdat de communicatie door verschillende tussenpersonen gaat. Er kan onderweg een heleboel misgaan.

Men kan niet altijd wie er toegang krijgt tot de systemen van het bedrijf. Een medewerker kan inloggen, maar zit deze medewerker wel echt aan de andere kant? Misschien werd het apparaat gestolen en kon die persoon zomaar inloggen.

Hoe kan er voorkomen worden dat een hacker het verkeer tussen de externe medewerker en het bedrijfsnetwerk kan onderscheppen? Er moet ook rekening gehouden worden met een Man-in-the-Middle-aanval (MitM), waarbij een externe medewerker denkt dat hij verbonden is met het bedrijfsnetwerk, terwijl hij eigenlijk met een ander netwerk verbonden is.

Tenslotte moet er ook nagedacht worden over hoe endpoint aanvallen door malware bestreden worden. Als deze endpoint een aanval krijgt, mag dit zich niet verspreiden naar het hele bedrijfsnetwerk.

3.3 Iframes

3.3.1 Wat is een iframe?

Een iframe is een frame in een frame. Het is een onderdeel van een HTML-element waarmee je documenten, video's en interactieve media kan insluiten in een pagina. Op die manier kan een secundaire webpagina op de hoofdpagina weergegeven worden.

Met het iframe-element is het mogelijk om een stuk inhoud van andere bronnen op te nemen. Het kan deze inhoud overal op de pagina integreren, zonder dat ze in de structuur van de web lay-out moeten opgenomen worden.

Iframes kunnen de pagina wel vertragen en een veiligheidsrisico vormen. Het is daarom belangrijk om enkel de inhoud te gebruiken van een website die te vertrouwen is.

3.3.2 Hoe wordt een iframe gebruikt?

Een iframe is een HTML-element dat door de <iframe>-tag kan gebruikt worden. Hier kunnen verschillende attributen voor gespecificeerd worden:

Attribuut	Specificatie
src	De URL naar het gevraagde document
srcdoc	De HTML-content van de gevraagde pagina
width/height	De breedte/hoogte van de iframe
allow	Feature policy voor de iframe
allowfullscreen	Kan een cross-origin iframe de Payment Request API aanspreken?
loading	Moet de browser de iframe onmiddellijk inladen of moet het wachten op bepaalde voorwaarden?
name	Naam van de iframe
referrerpolicy	Welke referrer informatie moet er doorgestuurd worden wanneer een iframe opgehaald wordt
sandbox	Maakt het mogelijk om een extra set restricties op de content van de iframe te zetten

Tabel 3: Attributen voor iframe [12]

3.3.3 Communicatie tussen iframes

Het is makkelijk om berichten tussen de parent (de applicatie waar de iframe instaat) en de iframe te versturen. Hiervoor wordt vooral de postMessage-functie gebruikt. Een praktisch voorbeeld is hieronder terug te vinden.

Een bericht van de parent naar de iframe versturen

In de parent is de volgende code nodig om een bericht te versturen:

```
const myiframe = document.getElementById('myIframe');


myIframe.contentWindow.postMessage('message', '*');
```

In de iframe wordt er gewacht op het bericht:

```
window.onmessage = function(event){
    if (event.data == 'message') {
        console('Message received!');
    }
};
```

Een bericht van de iframe naar de parent

In de iframe wordt het bericht verzonden op deze manier:

```
window.top.postMessage('reply', '*')
```

En in de parent is de volgende code nodig om op het bericht te wachten:

```
window.onmessage = function(event){
    if (event.data == 'reply') {
        console('Reply received!');
    }
};
```

3.3.4 Security

Wanneer een iframe gebruikt wordt, komt de inhoud hiervan meestal van derden. Hierdoor is er een groter risico om een potentiële kwetsbaarheid in de applicatie toe te laten of om een slechte user experience te leveren. Gelukkig kunnen er specifieke features geblacklist of gewhitelist worden.

Het sandbox-attribuut

Een volledige lijst van sandboxing flags en hun doel:

Flag	Details
allow-forms	Form submission toestaan
allow-modals	De bron toestaan om een nieuwe modal vensters te openen
allow-orientation-lock	De bron toestaan om de scherm oriëntatie te vergrendelen
allow-pointer-lock	De bron toestaan om Pointer Lock APO te gebruiken
allow-popups	De bron toestaan om een nieuwe pop-up of tab te openen
allow-popups-to-escape-sandbox	De bron toestaan om een nieuw venster te openen die het sandboxing-attribuut niet zal overerven
allow-presentation	De bron toestaan om een presentatie sessie te starten
allow-same-origin	De bron toestaan om zijn eigen oorsprong te behouden
allow-scripts	De bron toestaan om scripts te runnen
allow-top-navigation	De bron toestaan om te navigeren in de top-level browsing context
allow-top-navigation-by-user-activation	De bron toestaan om te navigeren in de top-level browsing context, maar enkel als dit door de gebruiker werd geactiveerd

Tabel 4: Sandboxing flags [13]

Wanneer een leeg sandbox-attribuut gebruikt wordt, zal de iframe volledig gesandboxed worden. Dit betekent dat de JavaScript in de iframe niet wordt uitgevoerd en dat alle flags hierboven beperkt worden.

3.4 Technologieën voor de implementatie

3.4.1 Node.js

Node.js is een runtime-omgeving die het mogelijk maakt om JavaScript-code onafhankelijk van een webbrowser uit te voeren.

Node.js heeft enkele belangrijke features, waardoor er vaak gekozen wordt om hiermee te werken:

- Asynchroon en event-driven: Alle API's van de Node.js library zijn asynchroon. Dit betekent dat een Node.js-gebaseerde server niet wacht op een API om data terug te geven.
- Snel: Het uitvoeren van code gaat snel doordat het gebouwd is met de V8 JavaScript Engine van Google Chrome.
- Single-threaded maar hoge schaalbaarheid: Node.js gebruikt een single-threaded model met event-looping. Het event mechanisme helpt de server om te antwoorden op een niet-blokkerende manier. Het zorgt ervoor dat de server een hoge schaalbaarheid heeft tegenover traditionele servers, die gelimiteerde threads creëren om requests te behandelen. Node.js gebruikt een single-threaded programma en ditzelfde programma kan een service van een groter aantal requests voorzien dan traditionele servers zoals Apache.
- Geen buffering: De data wordt niet gebufferd, maar de applicatie zal de data uitgeven in chunks.
- Licentie: Node.js is vrijgegeven onder de MIT licentie.

Figuur 11: Logo Node.js [29]

3.4.2 Express.js

Express is een flexibel Node.js webapplicatie framework dat een robuuste set van functies biedt voor web en mobiele applicaties. Met een groot aantal HTTP methoden en middleware tot uw beschikking, is het creëren van een robuuste API snel en eenvoudig.

Figuur 12: Express.js [30]

Aangezien Express.js een framework van Node.js is, is het grootste deel van de code al geschreven voor programmeurs om er makkelijk mee te werken. Het is licht van gewicht en helpt om webapplicaties op de server-side te organiseren in een meer georganiseerde MVC-architectuur. Express is een onderdeel van een op JavaScript gebaseerde technologie, genaamd MEAN software stack. Dit staat voor MongoDB, ExpressJS, AngularJS en Node.js. Hierbij is Express het backend deel van MEAN en beheert het routing, sessies, HTTP-verzoeken, foutafhandeling en meer.

Features van Express.js:

- Sneller server-side development: Express.js biedt veel veelgebruikte functies van Node.js in de vorm van functies die overal in het programma gemakkelijk kunnen worden gebruikt. Hierdoor wordt er niet meer urenlang gecodeerd.
- Middleware: Middleware is een deel van het programma dat toegang heeft tot de database, client verzoek en andere middlewares. Het is voornamelijk verantwoordelijk voor de systematisch organisatie van verschillende functies van Express.js
- Routing: ExpressJS biedt een zeer geavanceerd routing mechanisme dat helpt om de toestand van de webpagina te behouden met behulp van hun URL’s.
- Templating: ExpressJS biedt templating engines die de ontwikkelaars toelaten dynamische inhoud op de webpagina's te bouwen door HTML templates te bouwen op de server-side.
- Debugging: Debuggen is cruciaal voor de succesvolle ontwikkeling van web applicaties. ExpressJS maakt debugging makkelijker door een debugging mechanisme te bieden dat de mogelijkheid heeft om het exacte deel van de web applicatie te lokaliseren dat bugs bevat.

3.4.3 EJS

Embedded JavaScript templating of simpelweg EJS is een gemakkelijke templatingtaal waarmee HTML-opmaak kan gegenereerd worden met gewone JavaScript.

Enkele features:

- Snelle compilatie en rendering
- Eenvoudige template tags: <% %>
- Zowel server JS als browser ondersteuning
- Statisch cachen van tussenliggende JavaScript en sjablonen
- Voldoet aan het Express view-systeem

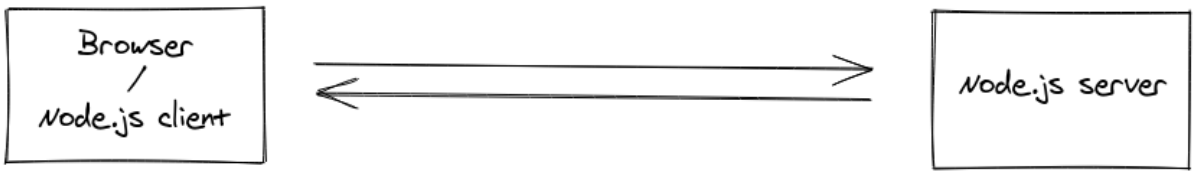
Wat is zo'n templatingtaal? Dit is een soort engine die het mogelijk maakt om statische template bestanden te gebruiken in een applicatie. Tijdens runtime vervangt de template engine variabelen in een template bestand door actuele waarden en transformeert de template in het HTML bestand dat naar de client wordt gestuurd. Deze benadering maakt het eenvoudiger om de HTML-pagina te ontwerpen.

3.4.4 Socket.io

Socket.io maakt real-time en event-based communicatie mogelijk. Het werkt op elk platform, browser of apparaat, met evenveel aandacht voor betrouwbaarheid als snelheid.

Figuur 13: Socket.io [31]

Socket.io bestaat uit een Node.js server en een JavaScript client library voor de browser die ook vanuit Node.js kan gedraaid worden.



Figuur 14: Socket.io [14]

Als dit mogelijk is, zal de client een WebSocket-connectie tot stand brengen en zal terugvallen op HTTP long polling als een connectie niet mogelijk is. WebSocket is een communicatie protocol die een full-duplex en “low latency”-kanaal tussen de server en de browser biedt.

Om Socket.io te kunnen gebruiken is het nodig om langs de client-side het `io()`-commando te gebruiken. Langs server-kant kan dit met `require("socket.io")`. Om een bericht te versturen via Socket.io wordt `socket.emit()` gebruikt en dit bericht opvangen gebeurt met `socket.on()`. Het eerste argument van deze commando’s bevat de naam van de connectie, gevolgd door de parameters die van backend naar frontend moeten gaan en omgekeerd. De volledige werking wordt verder besproken.

3.4.5 PeerJS

Figuur 15: PeerJS [15]

PeerJS omwikkelt de WebRTC-implementatie om een complete, configureerbare en eenvoudig te gebruiken peer-to-peer connectie-API te bieden. Uitgerust met een ID, kan een peer een peer-to-peer data of media stream verbinding maken met een remote peer.

Dit wordt gebruikt omdat rechtstreeks werken met WebRTC ingewikkeld kan zijn en PeerJS dit veel eenvoudiger maakt. Daarnaast biedt PeerJS ook een gratis cloud service die gebruikt zullen worden om de rooms te hosten.

3.4.6 RobotJS / RobotJS-browser

Figuur 16: RobotJS [16]

RobotJS is een desktop automation library die kan gebruikt worden voor het manipuleren van muis- en toetsenbordinvoer.

Belangrijk voor deze implementatie is vooral het bewegen van de muis en de mogelijkheid om te klikken. Deze functies kunnen in de documentatie van RobotJS teruggevonden worden als `mouseMove()` en `mouseClick()`.

Om de muis te bewegen, moet de juiste positie van de muis vastgesteld en doorgegeven worden. Deze positie (in x- en y-coördinaten) wordt dan meegegeven in de mouseMove()-functie. Dit is niet nodig om te klikken, aangezien de muis al op de juiste plaats staat. Er moet dus niets aan de functie meegegeven worden. Enkel de klik-actie moet vastgesteld zijn, waarop de actie in RobotJS wordt uitgevoerd.

RobotJS is bedoeld om in de server-kant te implementeren, maar in deze situatie is RobotJS nodig in de client-side. Daarom wordt RobotJS-browser gebruikt. De werking is gelijkaardig. Bijkomstig moet er een service gedraaid worden en moet er voor de functie window.robotjs.wrapper gebruikt worden.

RobotJS is een WIP (Work In Progress). Het is dus belangrijk om er rekening mee te houden dat de functies op elk moment kunnen aangepast worden.

3.5 Remote access in de browser implementeren

3.5.1 Installatie

Voor dit project zijn er verschillende dependencies nodig. Hiervoor wordt er gewerkt met npm. Dit is een soort pakketbeheerder die standaard een package.json-bestand aanmaakt door middel van het "`npm init -y`"-commando. Om een dependency te installeren wordt het "`npm i`"-commando gebruikt. Ik installeerde in eerste instantie Express, EJS, Socket.io, UUID en Nodemon. Verder in het document wordt er in detail verteld waar deze dependencies voor staan. Hieronder staan de gebruikte commando's nog eens opgelijst:

- `npm i express ejs socket.io`
- `npm i uuid`
- `npm i --save-dev nodemon`

Deze applicatie zal een server- en client-side hebben. Eerst wordt een server.js-bestand gemaakt. Deze is namelijk nodig voor de aanpassingen in het package.json-bestand. In package.json wordt de inhoud van "scripts" veranderd, zodat het overeenkomt met onderstaande code:

```
"scripts": {
  "devStart": "nodemon server.js"
},
```

Deze code zorgt ervoor dat bij het uitvoeren van het "`npm run devStart`"-commando, het server.js-bestand wordt uitgevoerd door middel van Nodemon. Nodemon is een tool die de node applicatie automatisch herstart wanneer server.js verandert. [17]

3.5.2 Basis code om de server gestart te krijgen

In server.js wordt de Express-server aangemaakt en wordt ervoor gezorgd dat Socket.io weet welke server het moet gebruiken. De server zal poort 3000 gebruiken.

```
const express = require('express');
const app = express();
const server = require('http').Server(app);
const io = require('socket.io')(server);

server.listen(3000);
```

Zoals te zien hierboven wordt "`require`" veel gebruikt in Node.js. De "`require`"-functie is een makkelijke manier om modules in de eigen applicatie op te nemen die uit afzonderlijke bestanden bestaan. De basis functionaliteit van "`require`" is dat het een JavaScript-bestand leest, uitvoert en dan een "`exports`"-object teruggeeft.

Er wordt een views-folder aangemaakt met hierin een room.ejs-bestand die basis HTML bevat:

```
<!DOCTYPE html>
<html>
<head>
    <title>Test</title>
</head>
<body>
    <h1>Room</h1>
</body>
</html>
```

De gebruikers moeten deze pagina natuurlijk ook kunnen zien. Daarom moet de juiste HTML-pagina meegegeven worden. Daarnaast krijgt elke gebruiker een eigen unieke roomId.

```
const { v4: uuidV4 } = require('uuid');

app.set('view engine', 'ejs');
app.use(express.static('public'));

app.get('/', (req, res) => {
    res.render('room', { roomId: uuidV4() });
});
```

Door de "`app.set`" weet de applicatie welke engine gebruikt wordt en zal er direct naar de views-folder gekeken worden bij het zoeken van een EJS-bestand. Met "`res.render`" wordt bepaald welk bestand in de views-folder gevraagd wordt. Het is ook mogelijk om parameters toe te voegen. In dit voorbeeld is roomId zo'n parameter. Deze roomId bestaat uit een UUID die via de uuid-package bekomen wordt.

Wanneer iemand verbinding maakt met de server zal deze in de volgende code terechtkomen. De code in "`connection`" zijn alle events waar de socket naar moet luisteren. Iemand toevoegen aan een room wordt als volgt gedaan:

```
io.on('connection', socket => {
    socket.on('join-room', (roomId, userId) => {
        console.log(roomId, userId);
    });
});
```

De roomId en userId zullen op dit moment nog "`undefined`" zijn. Daarom moeten er enkele scripttags in het EJS-bestand toegevoegd worden. Met onderstaande code kan de roomId doorgegeven worden. Daarnaast kan de frontend nu ook aan Socket.io en aan de eigen scriptfile. Deze laatste moet zelf nog aangemaakt worden.

```
<script> var ROOM_ID = "<%= roomId %>"</script>
<script src="/socket.io/socket.io.js"></script>
<script src="script.js"></script>
```

Het script.js-bestand wordt in een public-folder aangemaakt, omdat server.js naar JavaScript-bestanden in deze folder zal zoeken. Hieronder zal er al eens connectie gemaakt worden met de Socket.io backend. De roomId en userId zou te zien moeten zijn in de terminal. Op dit moment is de userId nog hardcoded (10), maar dit zal verder nog aangepast worden. De roomId zou wel moeten overeenkomen met de UUID die in de server werd doorgegeven aan het EJS-bestand.

```
const socket = io('/');


socket.emit('join-room', ROOM_ID, 10);
```

In het server-bestand moet de code aangepast worden zodat een room kan toegevoegd worden en iedereen in de room verwittigd wordt.

```
socket.on('join-room', (roomId, userId) => {
    socket.join(roomId);
    socket.to(roomId).emit('user-connected', userId);
});
```

De userId moet ook in script.js opgevangen worden. Voorlopig wordt er in deze functie enkel een “`console.log`”-uitgevoerd.

```
socket.on('user-connected', id => {
    console.log("User connected: " + id);
});
```

3.5.3 PeerJS

Zoals ik eerder aangaf, is de userId nog steeds hardcoded. Dit kan niet zo blijven. Hiervoor wordt PeerJS gebruikt. PeerJS zal elke gebruiker een ID geven, maar zal ook voor andere functies gebruikt worden in dit project. Om PeerJS te installeren, wordt het "`npm i -g peer`"-commando gebruikt. Om PeerJS te gebruiken, wordt "`peerjs --port 3001`" in de terminal getypt.

Ook voor PeerJS moet een scripttag een plaats in de EJS-file krijgen:

```
<script>
    var ROOM_ID = "<%= roomId %>"
</script>
<script src="https://unpkg.com/peerjs@1.3.1/dist/peerjs.min.js"></script>
<script src="/socket.io/socket.io.js"></script>
<script src="script.js"></script>
```

Door de scripttag is er ook in script.js toegang tot PeerJS. Om een peer te maken, wordt volgende code gebruikt. De eerste parameter is "`undefined`" omdat er dan een unieke ID aan de peer wordt gegeven door PeerJS. De poort moet overeenkomen met de poort die in het commando in de terminal wordt meegegeven.

```
const peer = new Peer(undefined, {
    host: '/',
    port: '3001'
});
```

Wanneer er een connectie is met de peer-server en deze een ID heeft teruggegeven, moet deze code (in script.js) uitgevoerd worden om als gebruiker te kunnen deelnemen in een room. ROOM_ID komt van de variabele die in het EJS-bestand werd gespecifieerd. De id wordt als parameter gegeven door PeerJS wanneer een peer open is.

```
peer.on('open', id => {
    socket.emit('join-room', ROOM_ID, id);
});
```

3.5.4 Betere interface

Nu er meerdere gebruikers aan een room kunnen toegevoegd worden, is het tijd om de interface gebruiksvriendelijker te maken. Hiervoor wordt het room.ejs-bestand aangepast:

```
<!DOCTYPE html>
<html>
<head>
  <title>Test</title>
</head>
<body>
  <h1>Room</h1>
  <p>UserId: <span id="userId"></span></p>

  <button id="makeRoom" onclick="makeRoom()">Make a new room</button>
  <input type="text" id="existingRoomId">
  <button id="joinRoom" onclick="joinRoom()">Join an existing room</button>

  <p>RoomId: <span id="roomId"></span></p>
  <button id="shareScreen" onclick="shareScreen()" style="display:
none;">Share my screen</button>
  <div id="screenDiv"></div>

    <script>var ROOM_ID = "<%= roomId %>"</script>
    <script
src="https://unpkg.com/peerjs@1.3.1/dist/peerjs.min.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="script.js"></script>
</body>
</html>
```

Nu is de userId te zien, maar is het ook mogelijk om een nieuwe room te maken. Daarnaast kan een roomId ingegeven worden om de gebruiker toe te voegen aan een bestaande room. Er is ook een knop om screensharing te starten. Als laatste is er een “div” die als placeholder fungeert. Wanneer er op “Share my screen” wordt geklikt, moet het scherm op de plaats van de placeholder te zien zijn.

Ook de script.js file moet aangepast worden zodat deze er als volgt uitziet:

```
const socket = io('/');
const peer = new Peer(undefined, {
    host: '/',
    port: '3001'
});
var userId;

peer.on('open', id => {
    document.getElementById("userId").innerHTML = id;
    userId = id;
});

socket.on('user-connected', id => {
    console.log("User connected: " + id);
});

function makeRoom(){
    document.getElementById("roomId").innerHTML = ROOM_ID;
    socket.emit('join-room', ROOM_ID, userId);
    document.getElementById("shareScreen").style.display = "block";
}

function joinRoom(){
    ROOM_ID = document.getElementById("existingRoomId").value;
    document.getElementById("roomId").innerHTML = ROOM_ID;
    socket.emit('join-room', ROOM_ID, userId);
    document.getElementById("shareScreen").style.display = "block";
}
function shareScreen(){

}
```

De juiste id wordt toegevoegd in de HTML. Daarnaast wordt het maken of lid worden van een room gedaan na het klikken op de juiste knoppen. Er staat ook al een functie die het klikken op de screenshare knop opvangt.

3.5.5 Nieuwe users moeten vorige users zien

Gebruikers kunnen toegevoegd worden aan een room. Echter worden de personen die voor de nieuwe deelnemer al in de room waren niet aan de nieuwe deelnemer getoond. Dit moet opgelost worden, startend door het script.js-bestand aan te passen:

```
var users = [];

socket.on('user-connected', id => {
    if (!users.includes(id)){
        console.log("User connected: " + id);
        users.push(id);
        socket.emit('user-connected', ROOM_ID, userId);
    }
});
```

Eerst wordt er gekeken of de gebruiker al bestaat in de users-array. Als dit niet het geval is, wordt de gebruiker hieraan toegevoegd. Daarnaast wordt de userId meegegeven aan de socket.

Nu wordt ook de server aangepast, zodat deze de inkomende user-connected krijgt.

```
io.on('connection', socket => {
    socket.on('join-room', (roomId, userId) => {
        socket.join(roomId);
        socket.to(roomId).emit('user-connected', userId);
    });

    socket.on('user-connected', (roomId, userId) => {
        socket.to(roomId).emit('user-connected', userId);
    });
});
```

Dezelfde user-connected in script.js wordt nu aangesproken vanuit de socket in de server.

3.5.6 Screensharing

Om de screensharing te starten moet de shareScreen()-functie in script.js aangepast worden.

```
function shareScreen(){
    const video = document.createElement("video");
    navigator.mediaDevices.getDisplayMedia({
        video: true
    }).then(stream => {
        video.srcObject = stream;
        video.addEventListener('loadedmetadata', () => {
            video.play();
        });
        document.getElementById("screenDiv").append(video);

        users.forEach(userId => {
            peer.call(userId, stream);
        });
        document.getElementById("shareScreen").style.display = "none";
    });
}
```

Hierbij wordt een video gemaakt en wordt de juiste stream meegegeven. Deze stream wordt bekomen met hulp van WebRTC. De video wordt dan op de juiste plaats getoond. Met peer.call worden de andere users op de hoogte gebracht van de stream.

De andere gebruikers moeten deze stream ook te zien krijgen. Dit wordt gedaan aan de hand van de volgende code:

```
peer.on('call', call => {
    call.answer();
    call.on('stream', userVideoStream => {
        document.getElementById("shareScreen").style.display = "none";
        const video = document.createElement("video");
        video.srcObject = userVideoStream;
        video.addEventListener('loadedmetadata', () => {
            video.play();
        });
        document.getElementById("screenDiv").append(video);
    });
});
```

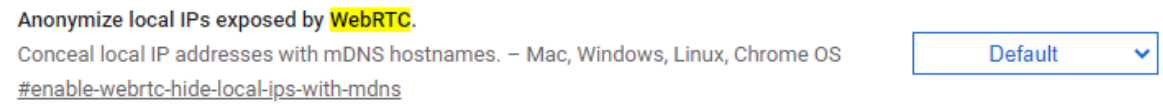
Eerst wordt de call beantwoord om vervolgens de video in de eigen HTML-pagina te zetten. Het beantwoorden van de call is zeer belangrijk. Als dit niet wordt gedaan, zal er geen video getoond worden.

3.5.7 Probleem met browsers

Wanneer deze applicatie lokaal gedraaid wordt, kunnen er problemen vastgesteld worden met de browsers. Op een bepaald moment is het nodig om te testen of de applicatie ook op andere computers (in hetzelfde netwerk) beschikbaar is. De applicatie blijkt beschikbaar, maar er is een probleem wanneer de screensharing moet getoond worden op de tweede computer. Dit probleem komt doordat WebRTC om de zoveel tijd het interne IP-adres van de computer die de applicatie draait, lekt. De browsers reageren hierop.

Om dit probleem op te lossen, wordt er gesurft naar "`chrome://flags`" en kan er gezocht worden naar "Anonymize local IPs exposed by WebRTC". Deze instelling staat op default. Standaard zal Chrome, maar ook andere browsers, deze instelling aanzetten. Als deze instelling wordt afgezet, is het probleem opgelost.

Wanneer deze proof of concept zou uitgewerkt worden, is het natuurlijk wel belangrijk om dit probleem nog eens te bekijken.



Figuur 17: Anonymize local IPs exposed by WebRTC (chrome://flags)

3.5.8 RobotJS

Om remote control mogelijk te maken, wordt er gebruikgemaakt van RobotJS. Hiermee kan de muis en het toetsenbord gecontroleerd worden. Om RobotJS te kunnen gebruiken, moet deze eerst geïnstalleerd worden met: `npm install robotjs`.

RobotJS kan niet direct in de browser gebruikt worden, daarom heb ik gebruikgemaakt van RobotJS-browser. Dit kon ik installeren door middel van het volgende commando: `npm install -g robotjs-browser`. RobotJS-browser moet natuurlijk ook luisteren naar binnenkomende requests. Dit gebeurt met het volgende commando: `robotjs-browser -h`.

Verder moet er in het EJS-bestand nog een scripttag toegevoegd worden met dit als resultaat:

```
<script> var ROOM_ID = "<%= roomId %>" </script>
<script src="https://unpkg.com/peerjs@1.3.1/dist/peerjs.min.js"></script>
<script src="/socket.io/socket.io.js"></script>
<script id="robotjs-script" type="text/javascript"
src="http://127.0.0.1:7569/robotjs-iframe-polyfill.js"></script>
<script src="script.js"></script>
```

Eerst moet elke beweging van de muis overgenomen worden. Hiervoor is er een EventListener nodig in het script.js-bestand. De positie van de muis wordt bepaald en meegegeven aan de socket.

```
document.getElementById('screenDiv').addEventListener("mousemove", function
(e) {
    var x = e.pageX - this.offsetLeft;
    var y = e.pageY - this.offsetTop;

    socket.emit("mouse-move", x, y, ROOM_ID);
});
```

Deze code wordt opgevangen in de server door middel van de volgende code in de connection. Elke gebruiker krijgt dus de positie van de muis.

```
socket.on('mouse-move', (x, y, roomId) => {
    socket.to(roomId).emit('mouse-move', x, y);
})
```

In script.js wordt dan nog de code om de actie uit te voeren, toegevoegd. De `moveMouse()`-functie zal nu door RobotJS uitgevoerd worden.

```
socket.on('mouse-move', (x, y) => {
    window.robotjs.wrapper.moveMouse(x, y);
});
```

Als de applicatie nu herladen wordt, is het dus mogelijk om de muis te bewegen en elke peer zou dit nu ook moeten zien.

Om ergens op te klikken, wordt dezelfde logica gehanteerd. Dus eerst wordt de EventListener weer toegevoegd in script.js:

```
document.getElementById('screenDiv').addEventListener("click", function ()
{
    socket.emit("mouse-click", ROOM_ID);
});
```

Dan wordt deze code opgevangen in server.js, zodat de andere gebruikers weten dat iemand heeft geklikt.

```
socket.on('mouse-click', (roomId) => {
    socket.to(roomId).emit('mouse-click');
});
```

En deze actie moet uitgevoerd worden in script.js. Aangezien de muis al op de juiste positie staat, is het hier niet meer nodig om nog een positie mee te geven. Het klikken zal dus gebeuren met dank aan RobotJS.

```
socket.on('mouse-click', () => {
    window.robotjs.wrapper.mouseClick();
});
```

De volledige code is zowel terug te vinden in de bijlagen als op de Gitlab-repository die in de bijlagen vermeld staat. Ook een filmpje met een volledige demo van deze applicatie is daar terug te vinden.

4 Resultaten

4.1 Technologieën

Node.js is een runtime-omgeving die zorgt voor de basis van deze proof of concept. Door Node.js kon er gebruikgemaakt worden van Express.js, een flexibel webapplicatie framework dat een robuuste set van functies biedt voor webapplicaties. Met Express kon er makkelijk een server opgezet worden en konden andere packages makkelijk gebruikt worden.

Express.js maakte het op zijn beurt mogelijk om EJS te gebruiken. Dit is een gemakkelijke templatingtaal waarmee HTML-opmaak kan gegenereerd worden met gewone JavaScript. Het uitzicht van de website is dus grotendeels hiermee gemaakt en het was daarnaast ook mogelijk om code vanuit de server naar de client te krijgen.

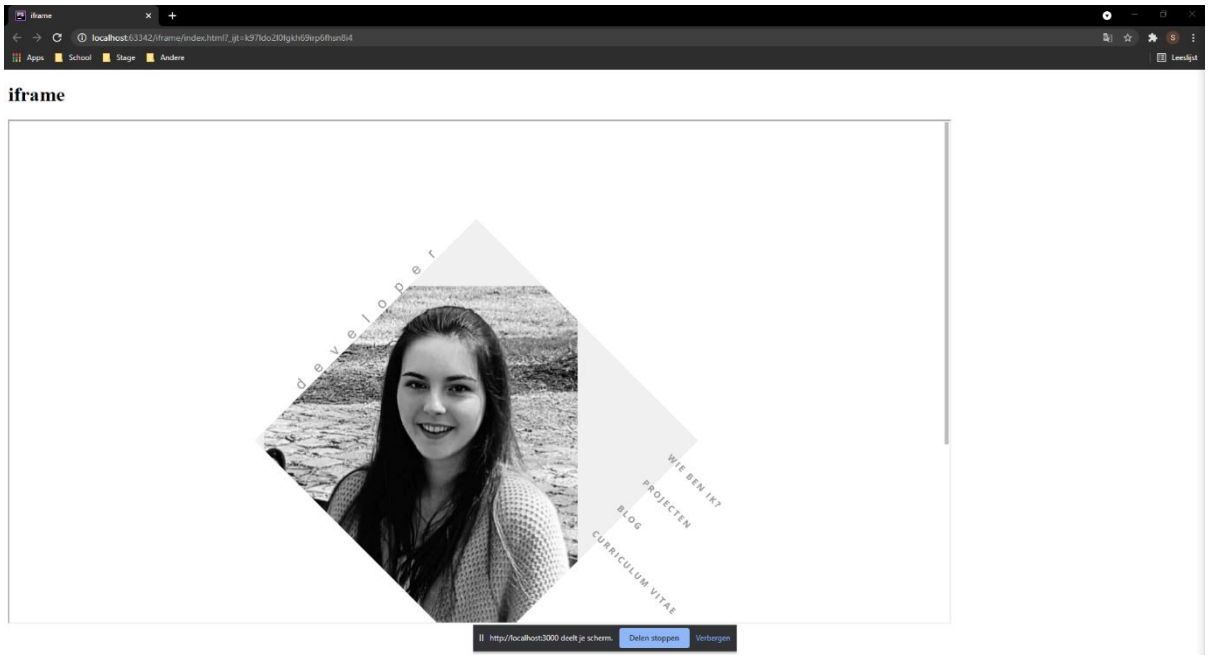
Socket.io maakt real-time en event-based communicatie mogelijk. Het bestaat uit een Node.js server en een JavaScript client library. De verbindingen in de proof of concept werden via Socket.io gedaan.

PeerJS omwikkelt de WebRTC-implementatie om een complete, configureerbare en eenvoudig te gebruiken peer-to-peer connectie-API te bieden. Omdat het gebouwd is op WebRTC, kunnen mediastreams verstuurd en ontvangen worden.

RobotJS is een desktop automation library die kan gebruikt worden voor het manipuleren van muis- en toetsenbordinvoer. Omdat RobotJS enkel in de server-kant kan geïmplementeerd worden, werd RobotJS-browser gebruikt voor de client-side.

4.2 Voorbeeld

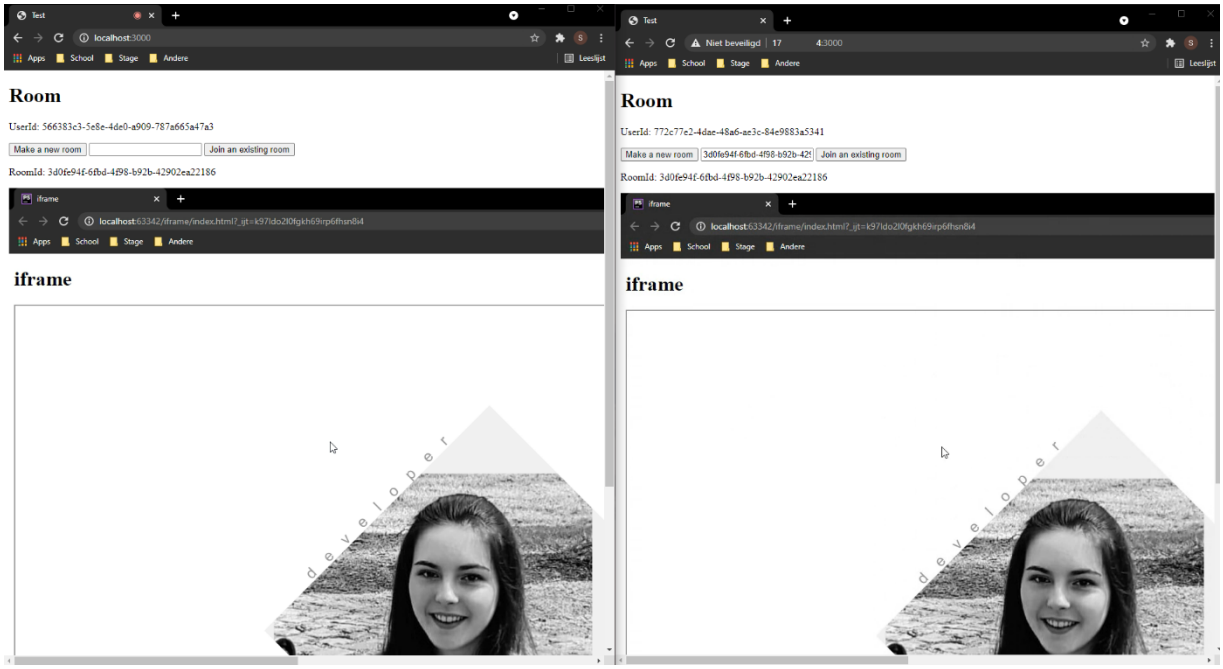
De afbeelding hieronder toont het scherm dat gedeeld wordt. Het gedeelde scherm bevat een website met een iframe in. De bron van de iframe is mijn eigen website. Deze werd gebruikt omdat ik hier zeker ben dat er enerzijds geen risicovolle inhoud in de website komt en anderzijds omdat mijn website de verbinding vanuit een iframe niet weigert.



Figuur 18: Het gedeelde scherm

Zoals hierboven te zien kan het delen op dit scherm gestopt worden.

De eerste helft van deze afbeelding toont de gebruiker die het scherm hierboven deelt. Deze persoon heeft de room ook gemaakt. Zowel de userId als de roomId is hierin te zien. Op de tweede helft wordt het scherm van de andere gebruiker getoond. Hier is ook te zien dat deze gebruiker de roomId die de eerste gebruiker kreeg zelf invult om in de juiste room terecht te komen.



Figuur 19: Scherm in de browser

De werking van de remote access control is hier niet te zien, omdat dit moeilijk is om op een afbeelding te tonen. Hiervoor heb ik een filmpje gemaakt. Hierin is te zien dat een gebruiker van op afstand een computer kan besturen door de muis te bewegen. Ook klikken is mogelijk, zelfs in de iframe.

De link naar het filmpje met de demo is terug te vinden in de bijlagen. En ook de bovenstaande afbeeldingen zijn in de bijlagen in een groter formaat te zien.

4.3 Wat moet er nog gebeuren?

Tijdens de proof of concept heb ik enkel lokaal gewerkt. Het is natuurlijk wel de bedoeling dat dit ook online werkt. Als het dan online staat, moeten er ook nog enkele testen uitgevoerd worden die ik niet kon doen omdat ik lokaal werkte. Zo wordt er best bekeken of het probleem met de “Anonymize local IPs exposed by WebRTC” opgelost is.

Een ander probleem door lokaal te werken, is het feit dat er op deze manier enkel een stream kan gegenereerd worden vanaf een localhost-connectie. Tijdens mijn onderzoek vond ik wel dat dit kan opgelost worden door een SSL-certificaat te maken. Dit is niet getest omdat dit niet tot de scope van mijn proof of concept behoorde.

Op dit moment is het mogelijk om de muis te bewegen en om te klikken, maar er zijn nog meer mogelijkheden met RobotJS. Zo kan er bijvoorbeeld ook toetsenbordinvoer geregistreerd en uitgevoerd worden. Vergeet hierbij niet dat RobotJS nog een WIP is. Het kan dus wel gebeuren dat niet alles even stabiel is en/of blijft.

Verder kan het bekijken van de security ook van belang zijn, aangezien remote access control wel risico’s met zich meebrengt.

5 Conclusie

Het is niet zo makkelijk om remote access control in de browser te implementeren. Er zijn verschillende technologieën voor nodig die allemaal moeten samenwerken om dit tot een werkend geheel te krijgen. Dit moet natuurlijk ook in een eigen project kunnen werken dus het is belangrijk om vooraf na te gaan of deze technologieën wel samenwerken met wat er in het eigen project al gebruikt wordt.

De meeste technologieën zijn makkelijk in gebruik en zijn logisch opgebouwd. Dat zorgt ervoor dat er makkelijk aanpassingen kunnen gemaakt worden. Zo kunnen developers die dit project willen gebruiken voor bijvoorbeeld enkel het bewegen van de muis, enkel deze functie nemen. Het is dan niet nodig om nog eens de klikfunctie te implementeren.

RobotJS doet al een groot deel van het werk, maar hiervan is er op dit moment geen stabiele release voor. Er moet dus altijd rekening gehouden worden met het feit dat deze package nog kan veranderen. Dit ligt natuurlijk niet in eigen handen, wat wel voor onzekerheden in het eigen project kan zorgen.

Zoals in de resultaten is meegegeven, zijn er wel nog wat werkpunten. Door RobotJS is er ook onzekerheid rond de stabiliteit. De proof of concept die hier beschreven staat, is zeker al een basis om op verder te bouwen. Het project kan nog verbeterd worden met de werkpunten die hier al beschreven staan. Als deze punten nog verbeterd worden, zal het implementeren van remote access control in een eigen applicatie in de browser zeker lukken.

6 Bronnen- en literatuurlijst

[1] Synoniemen.net, [Online]. Available: https://synoniemen.net/vertalingen.php?word=remote&source=en&target=nl. [Geopend 25 mei 2021].

[2] Wikipedia, „Codec,” 17 oktober 2016. [Online]. Available: https://nl.wikipedia.org/wiki/Codec. [Geopend 15 april 2021].

[3] J. C. Stephens, „Connection Tracking,” 5 april 2001. [Online]. Available: https://www.rigacci.org/wiki/lib/exe/fetch.php/doc/appunti/linux/sa/iptables/conntrack.html#:~:text=Connection%20tracking%20refers%20to%20the,this%20are%20known%20as%20stateful.. [Geopend 15 april 2021].

[4] Wikiversity, „Congestion Control Uses,” 3 december 2019. [Online]. Available: https://en.wikiversity.org/wiki/Congestion_control_uses#:~:text=Definition%3A%20Congestion%20control%20is%20a,is%20termed%20w%3Acongestive%20collapse.. [Geopend 15 april 2021].

[5] V. Saini, „Stream Multiplexing,” 15 december 2019. [Online]. Available: https://simpleaswater.com/stream-multiplexing/. [Geopend 15 april 2021].

[6] https://en.wikipedia.org/wiki/Flow_control_(data), „Flow Control (data),” 1 april 2021. [Online]. Available: https://en.wikipedia.org/wiki/Flow_control_(data). [Geopend 15 april 2021].

[7] Wikipedia, „Keepalive,” [Online]. Available: https://en.wikipedia.org/wiki/Keepalive. [Geopend 25 mei 2021].

[8] Wikipedia, „NAT traversal,” [Online]. Available: https://en.wikipedia.org/wiki/NAT_traversal. [Geopend 25 mei 2021].

[9] Wikipedia, „Head-of-line blocking,” [Online]. Available: https://en.wikipedia.org/wiki/Head-of-line_blocking. [Geopend 8 juni 2021].

[10] I. Grigorik, „High Performance Browser Networking,” 2013. [Online]. Available: https://hpbn.co/webrtc/. [Geopend 6 april 2021].

[11] M. Schmidt, „Remote Desktop Sessions,” 14 maart 2019. [Online]. Available: https://www.zumatech.com/wp-content/uploads/2019/03/rdp-blog-post-featured.png. [Geopend 24 mei 2021].

[12] W3schools, „HTML <iframe> Tag,” [Online]. Available: https://www.w3schools.com/tags/tag_iframe.ASP. [Geopend 19 mei 2021].

[13] N. Rifki, „The ultimate guide to iframes,” 23 januari 2020. [Online]. Available: https://blog.logrocket.com/the-ultimate-guide-to-iframes/. [Geopend 23 mei 2021].

[14] socket.io, „Socket.IO,” [Online]. Available: socket.io. [Geopend 28 april 2021].

[15] M. Lohr, „PeerChess – Chess with PeerJS,” 20 mei 2013. [Online]. Available: https://mlohr.com/wp-content/uploads/2013/05/peerjs_logo.png. [Geopend 24 mei 2021].

[16] RobotJS, „ROBOTJS,” [Online]. Available: http://robotjs.io/. [Geopend 24 mei 2021].

[17] npmjs, „nodemon,” [Online]. Available: https://www.npmjs.com/package/nodemon. [Geopend 26 mei 2021].

[18] L. Ben-David, „Build a Video Chat App with Socket.io, PeerJS, and Codesphere,” 27 maart 2021. [Online]. Available: https://javascript.plainenglish.io/building-a-video-chat-app-with-socket-io-peerjs-and-codesphere-949fbfe3a699. [Geopend 28 april 2021].

[19] Express, „Express - Node.js web application framework,” [Online]. Available: expressjs.com. [Geopend 28 april 2021].

[20] PeerJS, „PeerJS - Simple peer-to-peer with WebRTC,” [Online]. Available: peerjs.com. [Geopend 28 april 2021].

[21] tutorials point, „Node.js - Introduction,” [Online]. Available: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm#:~:text=is%20as%20follows%20%E2%88%92-,Node.,Node.. [Geopend 12 mei 2021].

[22] Wikipedia, „Node.js,” [Online]. Available: https://nl.wikipedia.org/wiki/Node.js. [Geopend 12 mei 2021].

[23] Soliton, „What is Remote Access Control? (No, it's not a stupid question),” 14 november 2019. [Online]. Available: https://blog.solitonsystems.com/it-security/what-is-remote-access-control. [Geopend 16 mei 2021].

[24] Besant Technologies, „What is Express.js?,” [Online]. Available: https://www.besanttechnologies.com/what-is-expressjs. [Geopend 16 mei 2021].

[25] M. Link, „robotjs-browser,” 14 april 2017. [Online]. Available: https://github.com/ml1nk/robotjs-browser. [Geopend 19 mei 2021].

[26] L. M., „What is an iFrame?,” 9 maart 2021. [Online]. Available: https://www.hostinger.com/tutorials/what-is-iframe/. [Geopend 19 mei 2021].

[27] EJS, „EJS,” [Online]. Available: https://ejs.co/. [Geopend 19 mei 2021].

[28] K. Devaliya, „Why use EJS template engine with ExpressJS instead of Handlebars?,” 22 september 2020. [Online]. Available: https://www.codementor.io/@kishandevaliya/why-use-ejs-template-engine-with-expressjs-instead-of-handlebars-15tc0hiefq. [Geopend 19 mei 2021].

[29] Wikipedia, „Node.js,” [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/d/d9/Node.js_logo.svg/1200px-Node.js_logo.svg.png. [Geopend 24 mei 2021].

[30] ButterCMS, „Express.js,” [Online]. Available: https://cdn.buttercms.com/2q5r816LTo2uE9j7Ntic. [Geopend 24 mei 2021].

[31] B. Gamble, „What's new in SocketIO 4?," 26 maart 2021. [Online]. Available: https://ik.imagekit.io/ably/ghost/prod/2021/03/socket-io-logo.jpeg?tr=w-1520. [Geopend 24 mei 2021].

[32] Wikipedia, „npm (software)," [Online]. Available: https://nl.wikipedia.org/wiki/Npm_(software). [Geopend 26 mei 2021].

[33] Web Dev Simplified, „How To Create A Video Chat App With WebRTC," 25 juli 2020. [Online]. Available: https://www.youtube.com/watch?v=DvlyzDZDEq4. [Geopend 10 april 2020].

[34] Codeboard Club, „Develop screen sharing application: Node JS App | Desktop sharing application development | Electron," 14 december 2020. [Online]. Available: https://www.youtube.com/watch?v=VQcG5LLxhGA. [Geopend 12 april 2020].

[35] Codeboard Club, „Develop remote desktop: Control PC from remote location | Node JS Project | Node JS & Express JS," 25 december 2020. [Online]. Available: https://www.youtube.com/watch?v=f2oaVXrKyuk. [Geopend 12 april 2020].

Overzicht van de bijlagen

Bijlage 1: Link naar GitLab en demo

De GitLab-repository is terug te vinden met de volgende link:
https://gitlab.com/SarahPolfliet/BAP

De link naar de demo is de volgende:
https://youtu.be/kdC6V_KTeN8

Bijlage 2: Code package.json

```
{
  "name": "Test",
  "version": "1.0.0",
  "description": "",
  "main": "script.js",
  "scripts": {
    "devStart": "nodemon server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "ejs": "^3.1.6",
    "express": "^4.17.1",
    "robotjs": "^0.6.0",
    "socket.io": "^4.0.1",
    "uuid": "^8.3.2"
  },
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
```

Bijlage 3: Code room.ejs

```
<!DOCTYPE html>
<html>
<head>
    <title>Test</title>
</head>
<body>
    <h1>Room</h1>
    <p>UserId: <span id="userId"></span></p>

    <button id="makeRoom" onclick="makeRoom()">Make a new room</button>

    <input type="text" id="existingRoomId">
    <button id="joinRoom" onclick="joinRoom()">Join an existing
room</button>

    <p>RoomId: <span id="roomId"></span></p>

    <button id="shareScreen" onclick="shareScreen()" style="display:
none;">Share my screen</button>
    <div id="screenDiv"></div>



    <script>
        var ROOM_ID = "<%= roomId %>"
    </script>
    <script
src="https://unpkg.com/peerjs@1.3.1/dist/peerjs.min.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script id="robotjs-script" type="text/javascript"
src="http://127.0.0.1:7569/robotjs-iframe-polyfill.js"></script>
    <script src="script.js"></script>
</body>
</html>
```

Bijlage 4: Code script.js

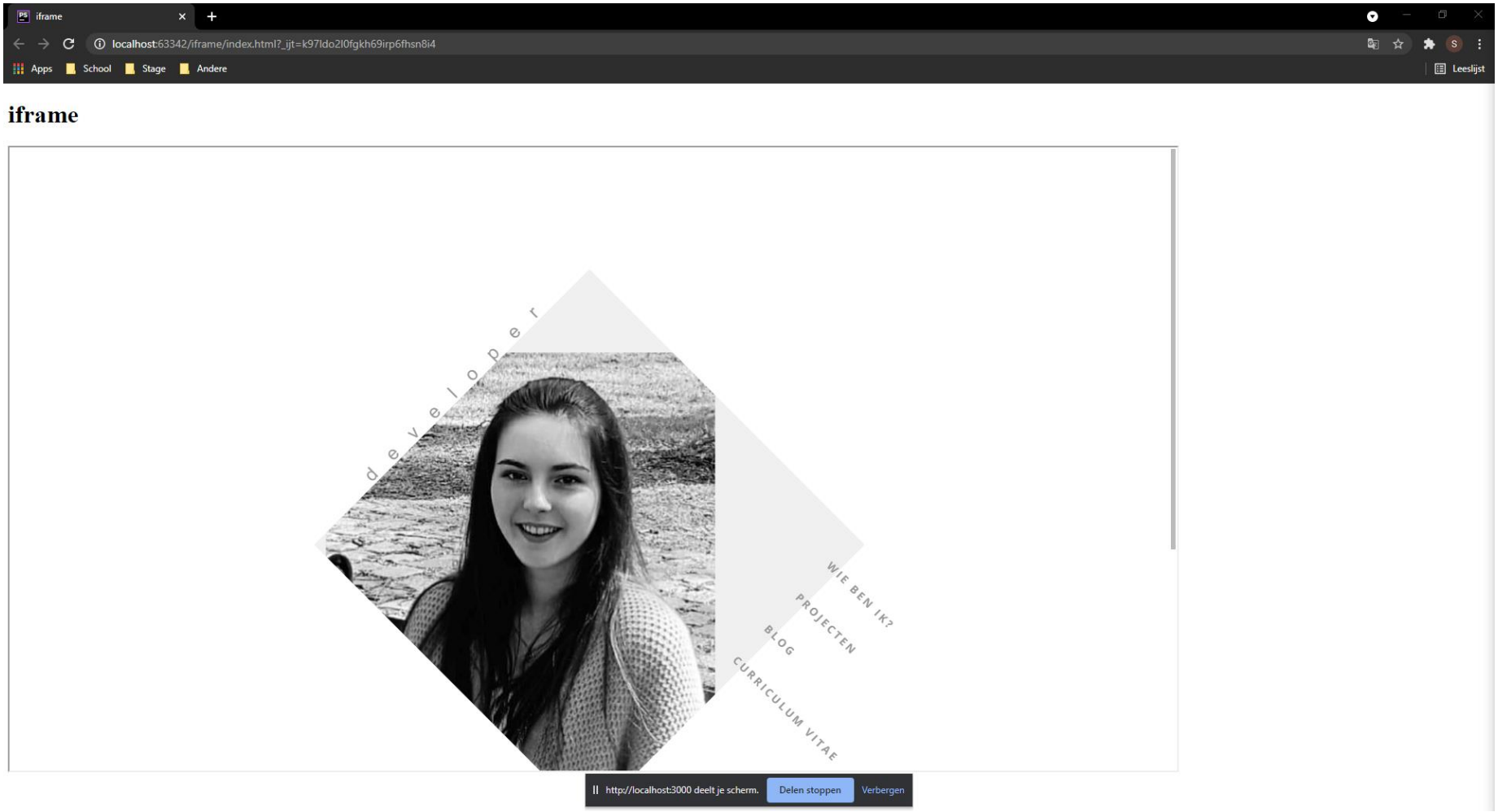
```
const socket = io('/');
const peer = new Peer(undefined, {
    host: '/',
    port: '3001'
});
var userId;
var users = [];

peer.on('open', id => {
    document.getElementById("userId").innerHTML = id;
    userId = id;
});

peer.on('call', call => {
    call.answer();
    call.on('stream', userVideoStream => {
        document.getElementById("shareScreen").style.display = "none";
        const video = document.createElement("video");
        video.id = "screen";
        video.srcObject = userVideoStream;
        video.addEventListener('loadedmetadata', () => {
            video.play();
        });
        document.getElementById("screenDiv").append(video);
    });
});


socket.on('user-connected', id => {
    if (!users.includes(id)){
        console.log("User connected: " + id);
        users.push(id);
        socket.emit('user-connected', ROOM_ID, userId);
    }
});


socket.on('mouse-move', (x, y) => {
    window.robotjs.wrapper.moveMouse(x, y);
});
```

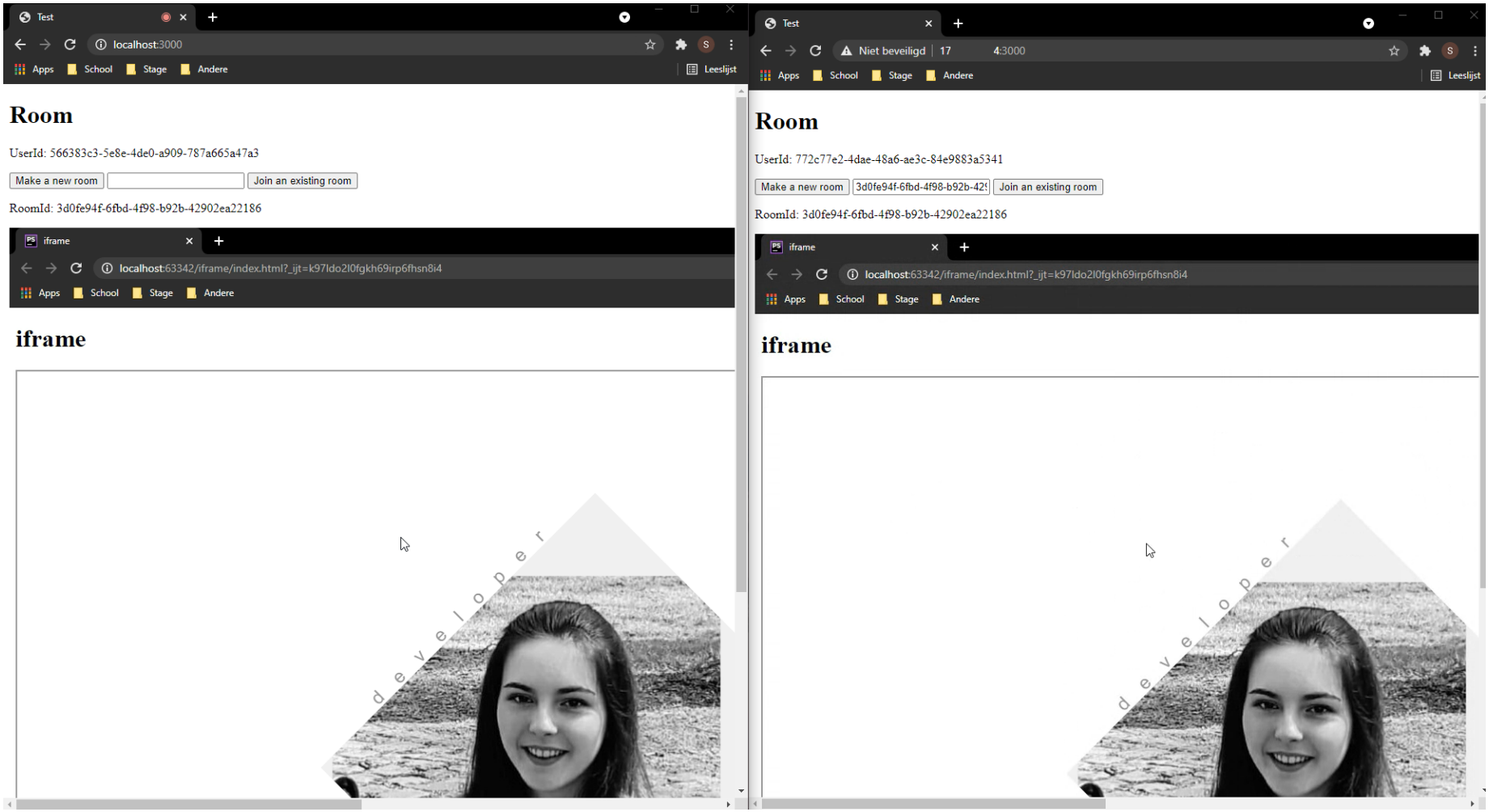
```
socket.on('mouse-click', () => {
    window.robotjs.wrapper.mouseClick();
});


function makeRoom(){
    document.getElementById("roomId").innerHTML = ROOM_ID;
    socket.emit('join-room', ROOM_ID, userId);
    document.getElementById("shareScreen").style.display = "block";
}

function joinRoom(){
    ROOM_ID = document.getElementById("existingRoomId").value;
    document.getElementById("roomId").innerHTML = ROOM_ID;
    socket.emit('join-room', ROOM_ID, userId);
    document.getElementById("shareScreen").style.display = "block";
}

function shareScreen(){
    const video = document.createElement("video");
    navigator.mediaDevices.getDisplayMedia({
        video: true
    }).then(stream => {
        video.srcObject = stream;
        video.addEventListener('loadedmetadata', () => {
            video.play();
        });
        document.getElementById("screenDiv").append(video);

        users.forEach(userId => {
            peer.call(userId, stream);
        });
        document.getElementById("shareScreen").style.display = "none";
    });
}
```

```
document.getElementById('screenDiv').addEventListener("mousemove", function
(e) {
    var x = e.pageX - this.offsetLeft;
    var y = e.pageY - this.offsetTop;

    socket.emit("mouse-move", x, y, ROOM_ID);
});

document.getElementById('screenDiv').addEventListener("click", function ()
{
    socket.emit("mouse-click", ROOM_ID);
});
```

Bijlage 5: Code server.js

```
const express = require('express');
const app = express();
const server = require('http').Server(app);
const io = require('socket.io')(server);
const { v4: uuidV4 } = require('uuid');

app.set('view engine', 'ejs');
app.use(express.static('public'));

app.get('/', (req, res) => {
    res.render('room', { roomId: 'room1' });
});

io.on('connection', socket => {
    socket.on('join-room', (roomId, userId) => {
        socket.join(roomId);
        socket.to(roomId).emit('user-connected', userId);
    });

    socket.on('user-connected', (roomId, userId) => {
        socket.to(roomId).emit('user-connected', userId);
    });

    socket.on('mouse-move', (x, y, roomId) => {
        socket.to(roomId).emit('mouse-move', x, y);
    });

    socket.on('mouse-click', (roomId) => {
        socket.to(roomId).emit('mouse-click');
    });
});

server.listen(3000);
```


iframe
localhost:63342/iframe/index.html?_ijt=k97ldo2l0fgkh69irp6fhsn8i4
Apps
School
Stage
Andere
Leeslijst
iframe
developer
WIE BEN IK?
PROJECTEN
BLOG
CURRICULUM VITAE
http://localhost:3000 deelt je scherm.
Delen stoppen
Verbergen


Test
localhost:3000
Apps School Stage Andere Leeslijst
Room
UserId: 566383c3-5e8e-4de0-a909-787a665a47a3
Make a new room
Join an existing room
RoomId: 3d0fe94f-6fbd-4f98-b92b-42902ea22186
iframe
localhost:63342/iframe/index.html?_ijt=k97ldo2l0fgkh69irp6fhsn8i4
Apps School Stage Andere
iframe
developer
Test
Niet beveiligd 17 4:3000
Apps School Stage Andere Leeslijst
Room
UserId: 772c77e2-4dae-48a6-ae3c-84e9883a5341
Make a new room
3d0fe94f-6fbd-4f98-b92b-42
Join an existing room
RoomId: 3d0fe94f-6fbd-4f98-b92b-42902ea22186
iframe
localhost:63342/iframe/index.html?_ijt=k97ldo2l0fgkh69irp6fhsn8i4
Apps School Stage Andere
iframe
developer