WebRTC för nyfikna

https://github.com/webrtc-for-the-curious/webrtc-for-the-curious

Contents

WebRTC för nyfikna

Den här boken har skrivits av WebRTC-utvecklare för att dela med sig av sina erfarenheter och kunskaper med världen. *WebRTC For The Curious* är en öppen källkodsbok skriven för de som alltid letar efter mer. Denna bok nöjer sig inte med abstraktioner.

Den här boken handlar om protokoll och API:er och inte om någon speciell programvara. Vi försöker sammanfatta RFC:er och få all odokumenterad kunskap samlad på ett ställe. Den här boken är ingen handledning och innehåller inte mycket kod.

WebRTC är en underbar teknik, men det är svår att använda. Denna bok är leverantörsagnostisk, och vi har försökt ta bort alla intressekonflikter.

Vem är den här boken för.

- Utvecklare som inte ens vet vad WebRTC löser och vill lära sig mer.
- Någon som redan använder WebRTC, men vill veta mer utöver API: erna.
- Etablerade utvecklare som behöver hjälp med felsökning.
- WebRTC-implementerare som behöver förstå mer om en specifik del.

Skriven för att läsas flera gånger

Denna bok är utformad för att läsas flera gånger. Varje kapitel är fristående, så du kan hoppa till vilken del av boken som helst utan att känna dig vilsen.

Varje kapitel syftar till att besvara en enda fråga med tre informationsnivåer.

- Vad behöver lösas?
- Hur löser vi det? Med tekniska detaljer om lösningen.

- Vart ska du lära dig mer?

Inget kapitel kräver några förkunskaper. Du kan börja var som helst i boken och börja lära dig. Denna bok kommer också att rekommendera resurser där man kan lära sig mer. Andra böcker täcker enskilda ämnen i mycket större djup. Denna bok syftar till att lära dig hela systemet, på bekostnad av expertkunskaper om de olika delarna.

Fritt tillgängligt och respekterar integritet

Denna bok finns på GitHub och WebRTCforTheCurious.com. Den är licensierad så att du kan använda den på det sätt du tycker är bäst. Du kan också ladda ner boken i sin nuvarande version som en ePub eller PDF på Svenska.

Denna bok är skriven av individer, för individer. Den är helt oberoende av leverantörer, så den kommer vi inte att ge några rekommendationer som kan vara en intressekonflikt.

Webbplatsen använder inte analys eller spårning.

Bli involverad!

Vi behöver din hjälp! Denna bok är helt utvecklad på GitHub och skrivs fortfarande. Vi uppmuntrar läsarna att öppna issues med frågor om saker som vi ännu inte gjort ett bra jobb med att beskriva.

Licens

Vad är WebRTC?

WebRTC, förkortning för Web Real-Time Communication, är både ett API och ett protokoll. WebRTC-protokollet är en uppsättning regler som två WebRTC-agenter använder för att sätta upp en säker tvåvägs realtidskommunikation. WebRTC API:et tillåter sedan utvecklare att använda WebRTC-protokollet. WebRTC API:et är endast specificerat för JavaScript.

Ett liknande förhållande skulle vara det mellan HTTP och Fetch API:et. WebRTC protokollet skulle vara HTTP, och WebRTC API:et motsvarar Fetch API:et.

WebRTC-protokollet finns tillgängligt i andra API:er och språk än JavaScript. Du kan också hitta servrar och domänspecifika verktyg för WebRTC. Alla dessa implementationer använder WebRTC-protokollet så att de kan kommunicera med varandra.

WebRTC-protokollet upprätthålls i IETF i arbetsgruppen rtcweb. WebRTC API:et är dokumenterat i W3C som webrtc.

Varför ska jag lära mig WebRTC?

Det här är en lista på några saker som WebRTC kommer att ge dig. Listan är inte komplett, bara exempel på saker. Oroa dig inte om du inte känner till alla dessa termer ännu, den här boken kommer att lära dig dem på vägen.

- Öppen standard
- Flera implementeringar
- Fungerar direkt i webbläsare
- Obligatorisk kryptering
- NAT Traversal
- Byggt på beprövad, befintlig, teknik
- Trängselkontroll
- Latens som kan mätas i bråkdelar av en sekund

WebRTC Protokollet är en samling av andra tekniker

Detta är ett ämne som kan ta en hel bok att förklara, men till att börja med delar vi upp det i fyra steg.

- Signalering
- Anslutning
- Säkerhet
- Kommunikation

Dessa fyra steg sker alltid i den här ordningen. Varje steg måste vara helt klart utan problem för att det efterföljande steget ska kunna börja.

En intressant sak med WebRTC är att varje steg faktiskt består av många andra protokoll och befintliga tekniker. I den meningen är WebRTC egentligen bara en kombination av beprövad teknik som har funnits sedan början av 2000-talet.

Vi kommer att gå igenom alla dessa steg i detalj senare, men det är bra att först förstå dem på en hög nivå. Eftersom de är beroende av varandra kommer det att hjälpa dig när vi senare förklarar syftet med vart och ett av dessa steg.

Signalering: Hur parter hittar varandra i WebRTC

När en WebRTC-agent startar har den ingen aning om vem den ska kommunicera med och vad de ska kommunicera om. Signalering löser problemet! Signalering används för att starta samtalet så att två WebRTC-agenter kan börja kommunicera.

Signalering använder ett befintligt protokoll SDP (Session Description Protocol). SDP är ett enkelt textbaserat protokoll. Varje SDP-meddelande består av

nyckel/värdepar och innehåller en lista med "media sections". SDP:n som de två WebRTC agenterna utbyter innehåller detaljer som:

- IP-adresser och portar som agenten kan nås på (kandidater).
- Hur många ljud- och videospår agenten vill skicka.
- Vilka ljud- och video-kodek som varje agent stöder.
- Värden som används vid anslutning (`uFrag`/`uPwd`).
- Värden som används vid säkringen (certifikatfingeravtryck).

Observera att signalering vanligtvis sker i en separat kanal (out-of-band); applikationer använder i allmänhet inte WebRTC för att skicka signalmeddelanden. Vilken arkitektur som helst som är lämplig för att skicka meddelanden kan användas för att vidarebefordra SDP:erna mellan de anslutande parterna. Många applikationer använder sin befintliga infrastruktur (som REST-API:er, WebSocket-anslutningar eller autentiserande proxys) för att på enklaste sätt utbyta SDP:er mellan rätt klienter.

Anslutning och NAT Traversal med STUN/TURN

De två WebRTC-agenterna vet nu tillräckligt med detaljer för att försöka ansluta till varandra. WebRTC använder sedan en annan etablerad teknik som kallas ICE.

ICE (Interactive Connectivity Establishment) är ett protokoll som skapades före WebRTC. ICE används för att upprätt av en anslutning mellan två agenter. Dessa agenter kan vara i samma nätverk eller på andra sidan världen. ICE är används för att skapa en direktanslutning utan att gå via en central server.

Den verkliga magin här är "NAT Traversal" och STUN/TURN-servrar. Dessa två begrepp är allt du behöver för att kommunicera med en ICE-agent i ett annat sub-nät. Vi kommer att utforska dessa ämnen djupare senare.

När ICE väl har anslutits går WebRTC vidare till att upprätta en krypterad transport. Denna kommunikationskanal används för ljud, video och data.

Säkra transportskiktet med DTLS och SRTP

Nu när vi har dubbelriktad kommunikation (via ICE) måste vi sätta upp en säker kommunikationskanal. Detta görs genom två protokoll som också är äldre än WebRTC. Det första protokollet är DTLS (Datagram Transport Layer Security) som helt enkelt är TLS över UDP. TLS är det kryptografiska protokollet som används för att säkra kommunikation via HTTPS. Det andra protokollet är SRTP (Secure Real-time Transport Protocol).

Först ansluter WebRTC genom att göra en DTLS-handskakning över anslutningen som upprättats av ICE. Till skillnad från HTTPS använder WebRTC inte en vanlig CA (Certificate Authority) för certifikatet. Istället hävdar WebRTC bara att certifikatet som utbyts via DTLS matchar. Denna DTLS-anslutning används sedan för DataChannel-meddelanden.

WebRTC använder sedan ett annat protokoll för ljud- och video-överföring som heter RTP. Vi skyddar våra RTP-paket med SRTP. Vi initierar vår SRTP-session genom att extrahera nycklarna från den förhandlade DTLS-sessionen. Vi kommer att gå igenom varför medieöverföring har ett eget protokoll i ett senare kapitel.

Nu är vi klara! Du har nu dubbelriktad och säker kommunikation. Om du har en stabil anslutning mellan dina WebRTC-agenter är det här all komplexitet du behöver. Tyvärr har den verkliga världen andra problem som paketförlust och brist på bandbredd. Nästa avsnitt handlar om hur vi hanterar dem.

Kommunicera med parter via RTP och SCTP

Vi har nu två WebRTC-agenter med säker dubbelriktad kommunikation. Låt oss börja kommunicera! Återigen använder vi två befintliga protokoll: RTP (Real-time Transport Protocol) och SCTP (Stream Control Transmission Protocol). Använd RTP för att utbyta media krypterat med SRTP och använd SCTP för att skicka och ta emot DataChannel-meddelanden krypterade med DTLS.

RTP är ganska minimalt, men ger oss allt vi behöver för att implementera realtidsströmning. Det viktiga är att RTP ger utvecklaren flexibilitet, så att de kan hantera latens, förlust och trängsel som de vill. Vi kommer att gå igenom detta ytterligare i mediekapitlet.

Det slutliga protokollet i stacken är SCTP. SCTP tillåter många leveransalternativ för meddelanden. Du kan till exempel välja att ha opålitlig leverans utan ordning, så att du kan få den latens som behövs för realtidssystem.

WebRTC, en samling protokoll

WebRTC löser väldigt många problem åt oss. Först kan detta verka väldigt ambitiöst, men det smarta med WebRTCs är att man inte försökte lösa allt själva. Istället återanvände man många befintliga tekniker för enskilda ändamål och kombinerade dem.

Detta gör att vi kan undersöka och lära oss om varje del individuellt, utan att bli överväldigade. Ett bra sätt att visualisera det hela är att en "WebRTC Agent" egentligen bara är en samordnare av många olika protokoll.

Hur fungerar API:et WebRTC

Det här avsnittet visar hur JavaScript API:et kopplas till protokollet. Detta är inte tänkt som ett omfattande demo av WebRTC API:et, utan mer för att skapa en mental modell för hur det hela hänger ihop. Om du inte känner till någon av dem är det ok. Det här kan vara ett roligt avsnitt att återvända till när du lär dig mer!

`new RTCPeerConnection` `RTCPeerConnection` är den högsta nivån av en "WebRTC Session". Den innehåller alla ovan nämnda protokoll. Delsystemen är alla

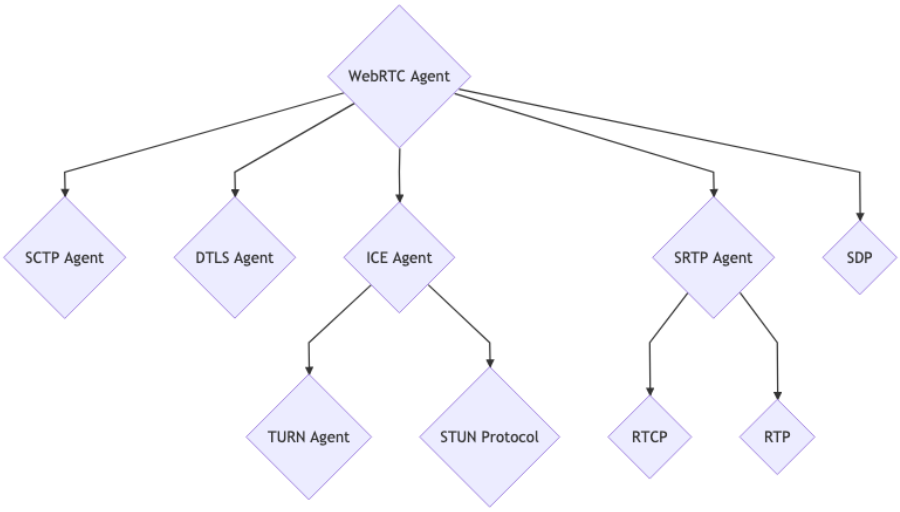


Figure 1: WebRTC Agent

allokerade men ingenting händer ännu.

`addTrack` `addTrack` skapar en ny RTP-ström. En slumpmässig synkroniseringskälla (SSRC) kommer att genereras för denna ström. Den här strömmen kommer sedan att finnas i Sessionsbeskrivningen som genereras av `createOffer` inuti en media-sektion. Varje samtal till `addTrack` skapar en ny SSRC och media-sektion.

Omedelbart efter att en SRTP-session har upprättats kommer dessa mediepaket att skickas via ICE efter att ha krypterats med SRTP.

`createDataChannel` `createDataChannel` skapar en ny SCTP-ström om det inte finns någon SCTP-koppling. Som standard är SCTP inte aktiverat men startas bara när en sida begär en datakanal.

Omedelbart efter att en DTLS-session har upprättats kommer SCTP-föreningen att skicka paket via ICE och krypteras med DTLS.

`createOffer` `createOffer` genererar en sessionsbeskrivning (Session Description) av det lokala tillståndet som ska delas med den andra parten.

Handlingen att anropa `createOffer` förändrar ingenting för den lokala parten.

`setLocalDescription` `setLocalDescription` gör alla begärda ändringar. `addTrack`, `createDataChannel` och liknande anrop är alla tillfälliga tills detta anrop görs. `setLocalDescription` anropas med det värde som genereras av `createOffer`.

Vanligtvis, efter det här anropet, kommer du att skicka erbjudandet till den andra parten, och de kommer att kalla `setRemoteDescription` med den.

`setRemoteDescription` `setRemoteDescription` är hur vi informerar den lokala agenten om andra kandidaters tillstånd. Så här görs signalering med JavaScript API:et.

När `setRemoteDescription` har anropats på båda sidor har WebRTC Agenterna nu tillräckligt med information för att börja kommunicera direkt part-till-part (P2P)!

`addIceCandidate` `addIceCandidate` tillåter en WebRTC-agent att lägga till fler ICE-kandidater på avstånd när de vill. Detta API skickar ICE-kandidaten direkt in i ICE-delsystemet och har ingen annan effekt på den större WebRTC-anslutningen.

`ontrack` `ontrack` är ett återanrop som avfyras när ett RTP-paket tas emot från den andra parten. De inkommande paketen skulle ha deklarerats i Sessionsbeskrivningen som skickades till `setRemoteDescription`.

WebRTC använder SSRC för att hitta rätt `MediaStream` och `MediaStreamTrack`, och avfyrar återanropet med dessa detaljer fyllda.

`oniceconnectionstatechange` `oniceconnectionstatechange` är en återanrop som avfyras för att visa tillståndet hos ICE-agenten. När du har fått en nätverksanslutning eller när du kopplas bort så får du det här meddelandet.

`onconnectionstatechange` `onconnectionstatechange` är en kombination av ICE Agenten och DTLS Agentens tillstånd. Du kan lyssna på detta meddelande för att få veta när uppsättningen av både ICE och DTLS har slutförts.

Vad är WebRTC-signalering?

När du skapar en WebRTC-agent vet den ingenting om den andra parten. Den har ingen aning om vem den kommer att ansluta till eller vad för data de ska skicka! Signalering är den första konfigureringen som gör ett samtal möjligt. Efter att dessa värden har utbytts kan WebRTC-agenterna kommunicera direkt med varandra.

Signalmeddelanden är bara vanlig text och WebRTC-agenterna bryr sig inte hur de skickas. De utbyts vanligtvis via Websockets, men det är inte ett krav.

Hur fungerar WebRTC-signalering?

WebRTC använder ett befintligt protokoll som kallas Session Description Protocol. Via detta protokoll delar de två WebRTC-agenterna alla tillstånd som krävs för att upprätta en anslutning. Protokollet i sig är enkelt att läsa och förstå. Komplexiteten kommer när man försöker förstå alla värden som WebRTC fyller i åt dig.

SDP protokollet är inte skapat specifikt för WebRTC. Eftersom WebRTC bara utnyttjar en liten del av protokollet ska vi bara gå igenom det vi behöver. När vi förstår protokollet kommer vi att gå vidare till hur det används i WebRTC.

Vad är *Session Description Protocol* (SDP)?

Sessionsbeskrivningsprotokollet definieras i RFC 8866. Det är ett nyckel/värde-protokoll med en ny rad efter varje värde, ungefär som en INI-fil. En sessionsbeskrivning (Session Description) innehåller noll eller fler mediebeskrivningar (Media Descriptions). Mentalt kan du se det som att en sessionsbeskrivning innehåller en uppsättning mediebeskrivningar.

En mediebeskrivning beskriver vanligtvis en enda ström av media. Så om du ville beskriva ett samtal med tre videoströmmar och två ljudspår skulle du ha fem mediebeskrivningar.

Hur man läser SDP

Varje rad i en sessionsbeskrivning börjar med ett enda tecken, det här är din nyckel. Det kommer sedan att följas av ett likhetstecken. Allt efter det lika tecknet är värdet. När värdet är klart kommer du att ha en ny rad.

Session Description Protocol definierar alla nycklar som är giltiga. Du kan bara använda bokstäver för nycklar enligt definitionen i protokollet. Dessa tangenter har alla betydande betydelse, vilket kommer att förklaras senare.

Här är ett exempel på en del av en sessionsbeskrivning:

```
a=my-sdp-value
a=second-value
```

Du har två rader. Var och en med nyckeln `a`. Den första raden har värdet `my-sdp-value`, den andra raden har värdet `second-value`.

WebRTC använder bara vissa SDP-nycklar

Inte alla nycklar som definieras i SDP används av WebRTC. Bara nycklar som används i JavaScript Session Establishment Protocol (JSEP), definierat i RFC 8829, är viktiga. Följande sju nycklar är de enda du behöver förstå just nu:

- `v` - Version, ska vara satt till `0`.

- `o` - Ursprung (Origin). Innehåller ett unikt ID som är användbart för omförhandlingar.
- `s` - Sessionsnamn, ska vara satt till `-`.
- `t` - Timing, ska vara satt till `0 0`.
- `m` - Mediebeskrivning (`m=<media> <port> <proto> <fmt> ...`), beskriven i detalj nedan.
- `a` - Attribut, ett fritextfält. Detta är den vanligaste raden i WebRTC.
- `c` - Anslutningsdata bör vara satt till `IN IP4 0.0.0.0`.

Mediebeskrivningar i en sessionsbeskrivning

En sessionsbeskrivning kan innehålla ett obegränsat antal mediebeskrivningar.

Varje mediabeskrivning kan innehålla en lista med olika format. Dessa format kopplas till RTP Payload Types. Den faktiska kodeken definieras sedan av ett attribut med värdet `rtpmap` i mediabeskrivningen. Vikten av RTP och RTP Payload Types diskuteras senare i kapitlet om Media. Varje mediebeskrivning kan innehålla ett obegränsat antal attribut.

Ta den här delen av en sessionsbeskrivning som ett exempel:

```
v=0
m=audio 4000 RTP/AVP 111
a=rtpmap:111 OPUS/48000/2
m=video 4000 RTP/AVP 96
a=rtpmap:96 VP8/90000
a=my-sdp-value
```

Du har två medibeskrivningar, en av typ av ljud med FMT `111` och en av typ av video med formatet `96`. Den första mediebeskrivningen har bara ett attribut. Detta attribut specificerar att Payload Type är `111` för Opus. Den andra mediebeskrivningen har två attribut. Det första attributet specificerar Payload Type är `96` för VP8, och det andra attributet är helt enkelt bara `my-sdp-value`.

Ett komplett exempel

Följande exempel innehåller alla begrepp som vi har pratat om. Det här är alla funktioner i SDP som WebRTC använder. Om du kan läsa och förstå detta, kan du läsa vilken WebRTC-session som helst!

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=-
c=IN IP4 127.0.0.1
t=0 0
m=audio 4000 RTP/AVP 111
a=rtpmap:111 OPUS/48000/2
m=video 4002 RTP/AVP 96
```

`a=rtpmap:96 VP8/90000`

- `v`, `o`, `s`, `c`, `t` är definierade, men de påverkar inte WebRTC-sessionen.
- Du har två mediebeskrivningar. En av typen `audio` (ljud) och en av typen `video`.
- Var och en av dessa har ett attribut. Detta attribut konfigurerar detaljer om RTP-pipelinen, vilket diskuteras i kapitlet "Mediekommunikation".

Hur *Session Description Protocol* och WebRTC fungerar tillsammans

Nästa pusselbit är att förstå *hur* WebRTC använder Session Description protokollet.

Vad är erbjudanden och svar?

WebRTC använder en erbjudande/svar-modell. Det betyder att den ena WebRTC-agenten gör ett "erbjudande" ("Offer") för att starta ett samtal, och den andra WebRTC-agenten ger ett "svar" ("Answer") om den är villig att acceptera det som erbjudits.

Detta ger svararen en chans att säga nej till kodeks i media-beskrivningarna (Media Descriptions). Det här är sättet två parter kommer överens om vilka format de är villiga att utbyta.

Transceivers är avsedda för sändning och mottagning

Transceivers är ett WebRTC-specifikt koncept som du kommer att se i API:et. Vad den gör är att exponera "Media Description" för JavaScript API. Varje mediebeskrivning blir en sändtagare. Varje gång du skapar en Transceiver läggs en ny mediebeskrivning till den lokala sessionsbeskrivningen.

Varje mediebeskrivning i WebRTC har ett riktningsattribut (direction attribute). Detta gör det möjligt för en WebRTC-agent att förklara "Jag ska skicka dig denna codec, men jag är inte villig att acceptera något tillbaka". Det finns fyra giltiga värden:

- `send`
- `recv`
- `sendrecv`
- `inactive`

SDP-värden som används av WebRTC

Nedan är en lista över några vanliga attribut som du kommer att se i en sessionsbeskrivning från en WebRTC-agent. Många av dessa värden styr delsystem som vi ännu inte har diskuterat.

`grupp:BUNDLE` Bundling är när man kör flera typer av trafik över en och samma anslutning. Vissa WebRTC-implementeringar använder en dedikerad anslutning per mediaström. Buntning är att föredra.

`fingerprint:sha-256` Detta är en hash av certifikatet som en part använder för DTLS. När DTLS-handskakningen är klar jämför du detta med det faktiska certifikatet för att bekräfta att du kommunicerar med den du har förväntat dig.

`setup:` Detta kontrollerar beteendet för DTLS-agenten. Detta avgör om det körs som en klient eller server efter att ICE har anslutit. De möjliga värdena är:

- `setup: active` - Kör som DTLS-klient.
- `setup: passive` - Kör som DTLS-server.
- `setup: actpass` - Be den andra WebRTC-agenten att välja.

`ice-ufrag` Detta är användarfragmentvärdet för ICE-agenten. Används för verifiering av ICE trafik.

`ice-pwd` Detta är lösenordet för ICE-agenten. Används för autentisering av ICE trafik.

`rtpmap` Detta värde används för att koppla en specifik kodek till en RTP Payload Type. Payload Types är inte statiska, så för varje samtal bestämmer avsändaren typen för varje kodek.

`fmtp` Definierar ytterligare värden för en Payload Type. Detta är användbart för att kommunicera en viss videoprofil eller kodekinställning.

`candidate` Detta är en ICE-kandidat som kommer från ICE-agenten. Det här är en möjlig adress som WebRTC-agenten är tillgänglig på. Dessa förklaras i mer detalj i nästa kapitel.

`ssrc` En synkroniseringskälla (SSRC) definierar ett enda mediaströmspår.

`label` är ID:t för den individuella strömmen. `mslabel` är ID:t för en container som kan innehålla flera strömmar.

Exempel på en WebRTC-sessionsbeskrivning

Följande är en fullständig sessionsbeskrivning genererad av en WebRTC-klient:

```
v=0
o=- 3546004397921447048 1596742744 IN IP4 0.0.0.0
s=-
t=0 0
a=fingerprint:sha-256 0F:74:31:25:CB:A2:13:EC:28:6F:6D:2C:61:FF:5D:C2:BC:B9:DB:3D:98:14:8D:1
```

```
a=group:BUNDLE 0 1
m=audio 9 UDP/TLS/RTP/SAVPF 111
c=IN IP4 0.0.0.0
a=setup:active
a=mid:0
a=ice-ufrag:CsxzEWmoKpJyscFj
a=ice-pwd:mktpbhgREmjEwUFSIJyPINPUhgDqJlSd
a=rtcp-mux
a=rtcp-rsize
a=rtpmap:111 opus/48000/2
a=fmtp:111 minptime=10;useinbandfec=1
a=ssrc:350842737 cname:yvKPspsHcYcwGFTw
a=ssrc:350842737 msid:yvKPspsHcYcwGFTw DfQnKjQQuwceLFdV
a=ssrc:350842737 mslabel:yvKPspsHcYcwGFTw
a=ssrc:350842737 label:DfQnKjQQuwceLFdV
a=msid:yvKPspsHcYcwGFTw DfQnKjQQuwceLFdV
a=sendrecv
a=candidate:foundation 1 udp 2130706431 192.168.1.1 53165 typ host generation 0
a=candidate:foundation 2 udp 2130706431 192.168.1.1 53165 typ host generation 0
a=candidate:foundation 1 udp 1694498815 1.2.3.4 57336 typ srflx raddr 0.0.0.0 rport 57336 ge
a=candidate:foundation 2 udp 1694498815 1.2.3.4 57336 typ srflx raddr 0.0.0.0 rport 57336 ge
a=end-of-candidates
m=video 9 UDP/TLS/RTP/SAVPF 96
c=IN IP4 0.0.0.0
a=setup:active
a=mid:1
a=ice-ufrag:CsxzEWmoKpJyscFj
a=ice-pwd:mktpbhgREmjEwUFSIJyPINPUhgDqJlSd
a=rtcp-mux
a=rtcp-rsize
a=rtpmap:96 VP8/90000
a=ssrc:2180035812 cname:XHbOTNRFnLtesHwJ
a=ssrc:2180035812 msid:XHbOTNRFnLtesHwJ JgtwEhBWNEiOnhuW
a=ssrc:2180035812 mslabel:XHbOTNRFnLtesHwJ
a=ssrc:2180035812 label:JgtwEhBWNEiOnhuW
a=msid:XHbOTNRFnLtesHwJ JgtwEhBWNEiOnhuW
a=sendrecv
```

Det här är vad vi får reda på från det här meddelandet:

- Vi har två mediasektioner, en för ljud och en för video.
- Båda är `sendrecv` transceiver. Vi får två strömmar och vi kan skicka tillbaka två.
- Vi har ICE-kandidater och autentiseringsdetaljer, så vi kan försöka ansluta.
- Vi har ett certifikatfingeravtryck, så vi kan ha ett säkert samtal.

Ytterligare ämnen

I senare versioner av denna bok kommer även följande ämnen att behandlas:

- Renegotiation
- Simulcast

Varför behöver WebRTC ett särskilt delsystem för anslutning?

De flesta applikationer som distribueras idag upprättar klient/serveranslutningar. En klient/serveranslutning kräver att servern har en känd och publik adress. En klient ansluter till en server, och servern svarar.

WebRTC använder inte en klient/server-modell utan skapar peer-to-peer (P2P) anslutningar. I en P2P-anslutning fördelas uppgiften att skapa en anslutning lika till båda parterna. Detta beror på att en publik nätverksadress (IP och port) är inget man kan förvänta sig i WebRTC, och den kan till och med ändras mitt under sessionen. WebRTC kommer att samla så mycket information som det går, och sedan göra allt den kan för sätta upp en dubbelriktad kommunikationsväg mellan två WebRTC-agenter.

Det kan dock vara svårt att skapa peer-to-peer-anslutning. Dessa agenter kan finnas i olika nätverk utan direkt anslutning. I situationer där direkt anslutning finns kan du fortfarande ha andra problem. I vissa fall talar dina klienter inte samma nätverksprotokoll (UDP <-> TCP) eller kanske har olika IP-versioner (IPv4 <-> IPv6).

Trots dessa svårigheter med att skapa en P2P-anslutning får du fördelar jämfört med traditionell klient/server-teknik på grund av följande funktioner WebRTC erbjuder.

Reducerade bandbreddskostnader

Eftersom mediekommunikation sker direkt mellan klienterna behöver du inte betala för eller drifta en egen server för att vidarebefordra media.

Lägre latens

Kommunikationen går snabbare när den är direkt! När en användare måste köra allt via din server gör det överföringarna långsammare.

Säker E2E-kommunikation

Direkt kommunikation är säkrare. Eftersom användarnas trafik inte går via din server, behöver de inte ens lita på att du inte dekrypterar den.

Hur fungerar det?

Processen som beskrivs ovan kallas Interactive Connectivity Establishment (ICE). Ytterligare ett protokoll som är äldre än WebRTC.

ICE är ett protokoll som försöker hitta det bästa sättet att kommunicera mellan två ICE-agenter. Varje ICE-agent publicerar hur den kan nås, dessa kallas kandidater. En kandidat är i huvudsak en nätverksadress för agenten som den tror att den andra klienten kan nå. ICE bestämmer sedan det bästa hopkopplingen av klienterna.

Den faktiska ICE-processen beskrivs mer detaljerat senare i detta kapitel. För att förstå varför ICE existerar är det bra att förstå vilket nätverksproblem vi behöver hantera.

Nätverksbegränsningar

ICE handlar om att övervinna begränsningarna i dagens nätverk. Innan vi utforskar lösningen, låt oss prata om några vanliga problem.

Inte i samma nätverk

För det mesta kommer den andra WebRTC-agenten inte ens att vara i samma nätverk. Ett typiskt samtal är vanligtvis mellan två WebRTC-agenter i olika nätverk utan direktanslutning.

Nedan är en figur som visar två distinkta nätverk, anslutna via internet. I varje nätverk har du två parter.

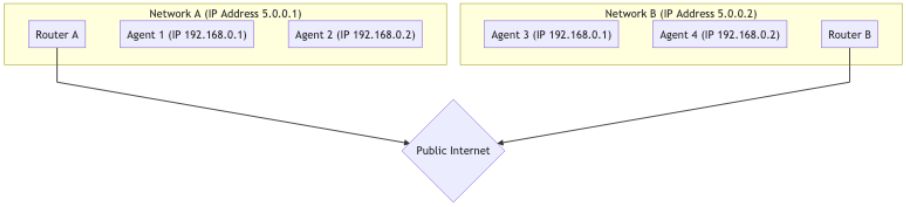


Figure 2: Två nätverk

För parterna i samma nätverk är det mycket enkelt att ansluta. Kommunikation mellan `192.168.0.1 -> 192.168.0.2` är lätt att göra! Dessa två parter kan ansluta till varandra utan någon extern hjälp.

En värd som använder `Router B` har dock inget sätt att komma åt något bakom "Router A". Hur kan den se skillnad på `192.168.0.1` bakom `Router A` och

samma IP bakom `Router B`? De är privata IP-adresser! En värd som använder `Router B` kan skicka trafik direkt till `Router A`, men anropet slutar där. Hur vet `Router A` vilken värd den ska vidarebefordra meddelandet till?

Protokollbegränsningar

Vissa nätverk tillåter inte UDP-trafik alls, eller kanske tillåter de inte TCP. Vissa nätverk kan ha en mycket låg MTU (Maximum Transmission Unit, storleken på varje paket). Det finns många variabler som nätverksadministratörer kan ändra som kan göra det svårt att kommunicera.

Brandvägg/IDS-regler

En annan är "Deep Packet Inspection" och andra intelligenta filtreringar. Vissa nätverksadministratörer kommer att köra programvara som försöker bearbeta varje paket. Många gånger förstår inte denna programvara WebRTC, så den blockerar den eftersom den inte vet vad den ska göra, t.ex. behandlas WebRTC-paket som misstänkta UDP-paket när de kommer på en godtycklig port som inte är känd.

NAT-kartläggning

NAT (Network Address Translation) är den magi som gör anslutningen av WebRTC möjlig. Så här tillåter WebRTC att två parter i helt olika subnät kommunicerar och behandlar problemet "inte i samma nätverk" ovan. Medan det skapar nya utmaningar, låt oss förklara hur NAT Mapping fungerar först.

Det använder inte ett relay, proxy eller server. Återigen har vi `Agent 1` och `Agent 2` och de finns i olika nätverk, men ändå flyter trafiken utan problem. Visualiserat ser det ut så här:

För att få denna kommunikation att upprätta skapar du en översättning med hjälp av NAT. Agent 1 använder port 7000 för att upprätta en WebRTC-anslutning med Agent 2. Detta skapar en koppling från `192.168.0.1:7000` till `5.0.0.1:7000`. Detta gör det möjligt för Agent 2 att nå Agent 1 genom att skicka paket till `5.0.0.1:7000`. Att skapa en NAT-mappning som i det här exemplet är som en automatisk version av att forward:a en port i din router.

Nackdelen med NAT är att det inte finns något konsekvent sätt att göra det på (t.ex. vidarebefordran av en statisk port), och att beteendet är olika i olika nätverk. Internetleverantörer och hårdvarutillverkare kan göra det på olika sätt. I vissa fall kan nätverksadministratörer till och med inaktivera det.

Den goda nyheten är att hela beteendet är förstått och observerbart, så en ICE-agent kan bekräfta att den har satt upp NAT rätt och vet då alla inställningarna i mappningen.

Dokumentet som beskriver dessa beteenden är RFC 4787.

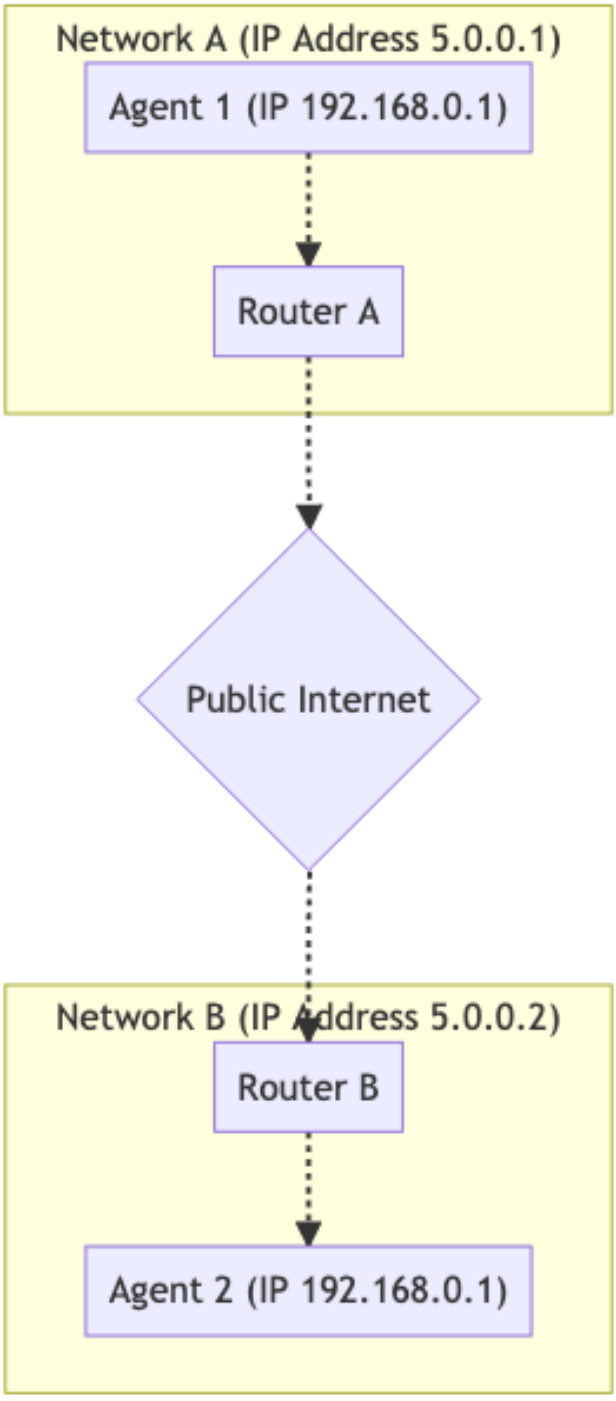


Figure 3: NAT mapping

Skapa en NAT-mappning

Att skapa en NAT-mappning är den enkla biten. När du skickar ett paket till en adress utanför ditt nätverk skapas en mappning! En NAT-mappning är bara en tillfällig offentlig IP och port som tilldelas av din NAT. Det utgående meddelandet kommer att skrivas om så att dess källadress ges av den nyligen mappade adressen. Om ett meddelande skickas tillbaka till den adressen, så dirigeras det automatiskt tillbaka till värden bakom den NAT:ade interna adressen. Detaljerna kring NAT är där det blir komplicerat.

Olika sorters NAT

En NAT tillhör en av tre olika kategorier:

Slutpunktoberoende mappning (Endpoint-Independent) En mappning skapas för varje avsändare i NAT:at nätverk. Om du skickar två paket till två olika adresser kommer NAT-mappningen att återanvändas. Båda parterna utanför det lokala nätet skulle se samma avsändare IP och port. Om de svarar, skulle det skickas tillbaka till samma lokala lyssnare.

Detta är det bästa fallet. För att ett samtal ska fungera, MÅSTE åtminstone en av parterna vara av den här typen.

Adressberoende mappning En ny mappning skapas varje gång du skickar ett paket till en ny adress. Om du skickar två paket till olika parter skapas två mappningar. Om du skickar två paket till samma part men till olika destinationsportar skapas INTE en ny mappning.

Adress- och port-beroende mappning En ny mappning skapas om fjärr-IP:t eller porten är annorlunda. Om du skickar två paket till samma IP utanför det lokala nätet, men olika portar, skapas en ny mappning för varje.

Mappning av filtreringsbeteenden

Mappningsfiltrering handlar om reglerna kring vem som får använda mappningen. Det finns tre liknande klassificeringar:

Slutpunktsoberoende filtrering Vem som helst kan använda mappningen. Du kan dela mappningen med flera andra parter och de kan alla skicka trafik till den.

Adressberoende filtrering Endast värden som mappningen skapades för kan använda mappningen. Om du skickar ett paket till agent `A` kan det svara tillbaka med så många paket som det vill. Om agent `B` försöker skicka ett paket till den mappningen ignoreras det.

Adress- och port-beroende filtrering Endast IP:t och porten för vilken mappningen skapades för kan använda den. Om du skickar ett paket till agent `A:5000` kan det svara tillbaka med så många paket som det vill. Om en agent `A:5001` försöker skicka ett paket till samma mappningen ignoreras paketet.

Uppdatera en mappning

Det rekommenderas att om en mappningen inte används på 5 minuter ska den förstöras. Detta är helt upp till ISP eller hårdvarutillverkaren.

STUN

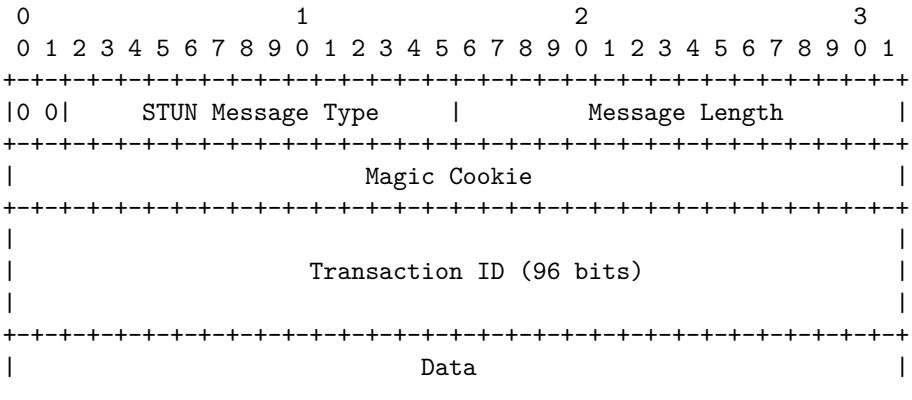
STUN (Session Traversal Utilities for NAT) är ett protokoll som skapades för att jobba med NATs. Det är ytterligare en teknik som skapats före WebRTC (och ICE!). Specifikationen hittar du i RFC 8489, som också definierar STUN-paketformatet. STUN-protokollet används också av ICE/TURN.

STUN är användbart eftersom det tillåter programmering av NAT mappning. Innan STUN kunde vi sätta upp ett NAT, men vi hade ingen aning om vilket IP och port det var! STUN ger dig inte bara möjligheten att skapa en NAT mappning, du får också detaljerna så att du kan dela dem med andra så att de kan skicka trafik till dig via den mappning du just skapade.

Låt oss börja med en grundläggande beskrivning av STUN. Senare kommer vi att utöka TURN- och ICE-användningen. För närvarande ska vi bara beskriva flödet för begäran / svar för att skapa en kartläggning. Sedan kommer vi att prata om hur man får detaljerna i den att dela med andra. Detta är processen som händer när du har en `stun:`-server i dina ICE-URL:er för en WebRTC PeerConnection. Kort sagt, STUN hjälper en server bakom ett NAT:at nätverk att lista ut vilken mappning som skapades genom att fråga en STUN-server utanför det lokala nätverket vad den kan se.

Protokollstruktur

Varje STUN-paket har följande format:



STUN Meddelande typ Varje STUN-paket har en typ. Just nu bryr vi oss bara om följande två:

- Bindningsförfrågan - `0x0001`
- Bindningssvar - `0x0101`

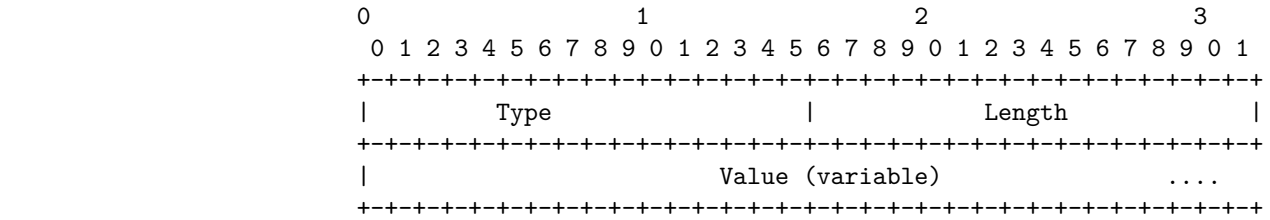
För att skapa en NAT-mappning gör vi ett `Binding Request`. Sedan svarar servern med ett `Binding Response`.

Meddelandets längd Hur långt avsnittet `Data` är. Detta avsnitt innehåller godtycklig data som definieras av `Message Type`.

Magic Cookie Det fasta värdet `0x2112A442` i nätverksbyteordning (network byte order), det hjälper till att skilja STUN-trafik från andra protokoll.

Transaktions ID En 96-bitars identifierare som unikt identifierar en begäran/svar. Detta hjälper dig att para ihop dina begäran och svar.

Data Data kommer att innehålla en lista med STUN-attribut. Ett STUN-attribut har följande struktur:



Ett `STUN Binding Request` använder inga attribut. Det betyder att en `STUN Binding Request` endast innehåller en header.

En `STUN Binding Response` använder en `XOR-MAPPED-ADDRESS (0x0020)`. Detta attribut innehåller en IP och port. Detta är IP och port för den NAT-mappning som skapas!

Skapa en NAT-mappning

Att skapa en NAT-mappning med STUN kräver bara att skicka en begäran! Du skickar ett `STUN Binding Request` till STUN-servern. STUN-servern svarar sedan med en `STUN Binding Response`. Denna `STUN Binding Response` kommer att innehålla en `Mapped Address`. Den "mappade adressen" är hur STUN-servern ser dig. Det är din "NAT-mappning". Den "mappade adressen" är vad du skulle dela om du vill att någon ska skicka paket till dig.

Din “mappade adress” kallas också för ditt “publika IP” eller ibland `Server Reflexive Candidate`.

Bestämma NAT-typ

Tyvärr kan en “mappad adress” kanske ändå inte vara användbar. Om reglerna är “adressberoende” (`Address Dependent`) kan bara STUN-servern skicka trafik till dig. Om du delade adressen och en annan agent försöker skicka meddelanden kommer de att ignoreras. Detta gör det värdelöst för att kommunicera med andra. Du kanske tycker att fallet “adressberoende” faktiskt går att lösa om STUN-servern också kan vidarebefordra paket åt dig till din andra part! Detta leder oss till en annan teknik, TURN.

RFC 5780 definierar en metod för att göra ett test för att lista ut din NAT-typ. Detta är användbart eftersom du kan veta i förväg om direktanslutning var möjlig.

TURN

TURN (Traversal Using Relays around NAT) definieras i [RFC 8656](https://tools.ietf.org/html/rfc8656) är lösningen när direktanslutning inte är möjlig. Det kan bero på att du har två NAT-typer som är oförenliga eller kanske inte kan tala samma protokoll! TURN kan också användas för sekretessändamål. Genom att köra all din kommunikation genom TURN döljer du klientens faktiska adress.

TURN använder en dedikerad server. Denna server fungerar som en proxy för en klient. Klienten ansluter till en TURN-server och skapar en allokering (`Allocation`). Genom att skapa en allokering får en klient en tillfällig IP/port/protokoll som kan användas för att skicka trafik tillbaka till klienten. Den här nya lyssnaren kallas en `Relayed Transport Address`. Tänk på det som en vidarebefordringsadress, du använder den så att andra kan skicka dig trafik via TURN! För varje klient du ger din `Relay Transport Address` till måste du skapa ett nytt “tillstånd” (`Permission`) för att tillåta kommunikation med dig.

När du skickar utgående trafik via TURN skickas den via din `Relayed Transport Address`. När en annan klient får trafik ser den ut att komma från TURN-servern.

TURN Livscykel

Följande är allt som en klient som vill skapa en TURN-allokering måste göra. Kommunikation med någon som använder TURN kräver inga ändringar. Den andra klienten får ett IP och en port, och de kommunicerar med den som vilken annan server som helst.

Tilldelningar (Allocations) Tilldelningar är kärnan i TURN. En "allokering" är i grunden en "TURN Session". För att skapa en TURN-allokering kommunicerar du med en TURN `Server Transport Address` (vanligtvis på port `3478`).

När du skapar en tilldelning måste du ange följande: * Användarnamn/lösenord - För att skapa TURN-tilldelningar krävs autentisering * Allokeringstransport - Transportprotokollet mellan servern (`Relayed Transport Address`) och klienterna. Kan vara UDP eller TCP. * Even-Port - Du kan begära sekventiella portar för flera tilldelningar, inte relevanta för WebRTC.

Om begäran lyckades får du svar med TURN-servern med följande STUN-attribut i Data-sektionen: * `XOR-MAPPED-ADRESS` - `Mappad adress` för din `TURN Client`. När någon skickar data till "Relayed Transport Address", vidarebefordras den till. * `RELAYED-ADRESS` - Det här är adressen som du ger ut till andra klienter. Om någon skickar ett paket till den här adressen vidarebefordras det till TURN-klienten. * `LIFETIME`- Hur länge tills denna TURN-fördelning tas bort. Du kan förlänga livslängden genom att skicka ett uppdaterings anrop (`Refresh`).

Behörigheter En klient kan inte skicka till din `Relayed Transport Address` förrän du skapar ett tillstånd för den. När du skapar en behörighet berättar du för TURN-servern att denna IP och Port får skicka inkommande trafik.

Klienten måste ge dig IP och port som den ser ut för TURN-servern. Det betyder att den ska skicka ett `STUN Binding Request` till TURN-servern. Ett vanligt felfall är att en klient skickar ett `STUN Binding Request` till en annan server. De kommer då att be dig skapa en behörighet för denna IP.

Låt oss säga att du vill skapa en behörighet (`permission`) för en klient bakom en "adressberoende mappning". Om du genererar en "mappad adress" från en annan TURN-server kommer all inkommande trafik att ignoreras. Varje gång de kommunicerar med en annan klient genererar den en ny mappning. Behörigheterna löper ut efter 5 minuter om de inte uppdateras.

SendIndication/ChannelData Dessa två meddelanden är avsedda för TURN-klienten att skicka meddelanden till en klient.

SendIndication är ett fristående meddelande. Det innehåller den information du vill skicka, och vem du vill skicka den till. Detta är slösaktigt om du skickar många meddelanden till en annan klient. Om du skickar 1000 meddelanden kommer du att upprepa deras IP-adress 1000 gånger!

ChannelData låter dig skicka data, utan att upprepa IP-adressen. Du skapar en kanal med en IP och port. Du skickar sedan med ChannelId och IP och port fylls i på serversidan. Detta är bättre om du skickar många av meddelanden.

Refreshing Allokeringar kommer att ta bort sig själva automatiskt. TURN-klienten måste uppdatera dem innan deras `LIFETIME` som ges när tilldelningen

skapas går ut.

TURN Användning

TURN Användningen finns i två former. Vanligtvis har du en part som fungerar som en "TURN klient" och den andra sidan kommunicerar direkt. I vissa fall kan du ha TURN-server på båda sidor, till exempel för att båda klienterna finns i nätverk som blockerar UDP, och därför sker anslutningen till respektive TURN-servrar via TCP.

Dessa figurer hjälper till att illustrera hur det skulle se ut.

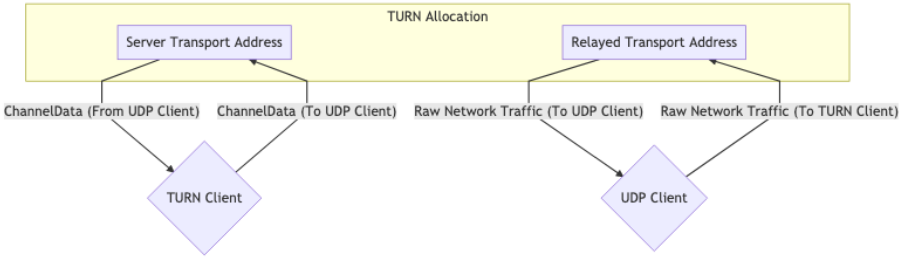


Figure 4: En TURN allokering

En TURN allokering för kommunikation

Två TURN allokeringar för kommunikation

ICE

ICE (Interactive Connectivity Establishment) är tekniken WebRTC använder för att ansluta två klienter med varandra. Definierad i RFC 8445, detta är ytterligare en standard som är återanvänd i WebRTC. ICE är ett protokoll för att sätta upp av anslutningar. Det hittar alla möjliga rutter (routes) mellan de två klienterna och säkerställer att de kan kommunicera med varandra.

Dessa rutter är kända som `Candidate Pairs`, vilket är en anslutning mellan en lokal och en extern adress. Det är här STUN och TURN används tillsammans med ICE. Dessa adresser kan vara din lokala IP-adress plus en port, "NAT-mappning" eller en `Relayed Transport Address`. Båda klienterna listar alla adresser de vill använda, skickar över dem och försöker sedan ansluta.

Två ICE-agenter kommunicerar med hjälp av ICE-ping-paket (`connectivity checks`) för att sätta upp en anslutning. När anslutningen har upprättats kan

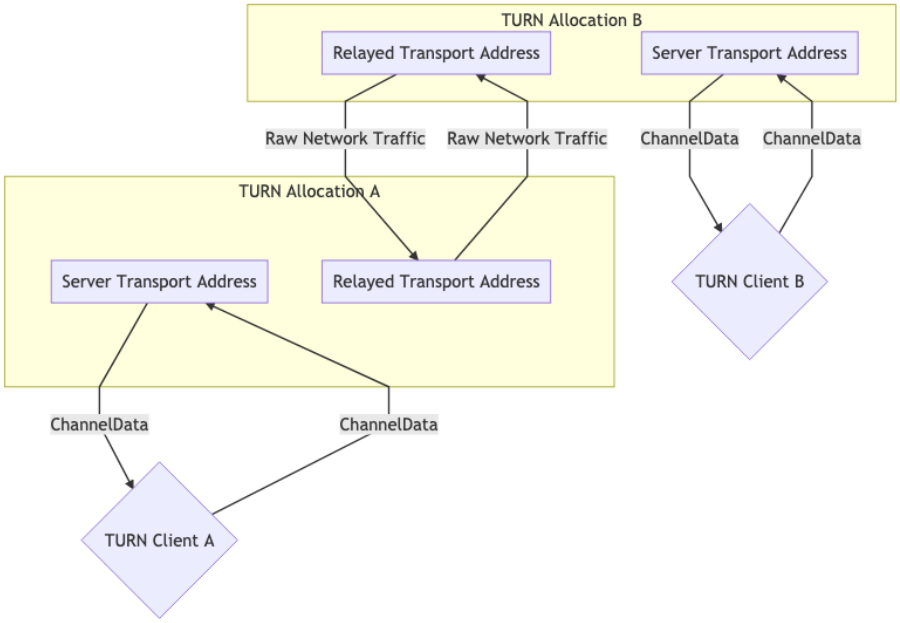


Figure 5: Två TURN allokeringar

de skicka vilken data de vill. Det blir som att använda en normal socket. Dessa kontroller använder STUN-protokollet.

Skapa en ICE-agent

En ICE-agent är antingen `Controlling` eller `Controlled`. Den "kontrollerande" agenten är den som bestämmer vilken kandidat (`Candidate Pair`) som ska användas. Vanligtvis är den klient som skickar erbjudandet den kontrollerande sidan.

Varje klient måste ha ett "användarfragment" (`user fragment`) och ett "lösenord" (`password`). Dessa två värden måste utbytas innan anslutningskontroller kan göras. "Användarfragmentet" skickas i klartext och är användbart för att separera (demuxing) flera ICE-sessioner. Lösenordet används för att generera attributet `MESSAGE-INTEGRITY`. I slutet av varje STUN-paket finns det ett attribut som är en hash för hela paketet med "lösenordet" som en nyckel. Detta används för att autentisera paketet och se till att det inte har blivit manipulerat.

För WebRTC skickas alla dessa värden via "sessionsbeskrivningen" som beskrevs i förra kapitlet.

Kandidatsamling

Vi måste nu samla alla möjliga adresser vi kan nås på. Dessa adresser kallas kandidater.

Klient En klientkandidat lyssnar direkt på ett lokalt nätverksinterface, antingen via UDP eller TCP.

mDNS En mDNS-kandidat liknar en klientkandidat, men IP-adressen är dold. Istället för att informera den andra sidan om din IP-adress, ger du dem ett UUID som värdnamn. Du ställer istället upp en multicast-lyssnare och svarar om någon frågar efter det UUID som du publicerat.

Om du är i samma nätverk som agenten kan du hitta varandra via multicast. Om du inte är i samma nätverk kommer du inte att kunna ansluta (om inte nätverksadministratören uttryckligen konfigurerat nätverket så att multicast-paket kan passera).

Detta är användbart för sekretessändamål. En användare kan ta reda på din lokala IP-adress via WebRTC med en klientkandidat (utan att ens försöka ansluta till dig), men med en mDNS-kandidat får de nu bara ett slumpmässigt UUID.

Serverreflexiv kandidat En serverreflexiv kandidat genereras genom att göra ett `STUN Binding Request` till en STUN-server.

När du får en `STUN Binding Response`, är `XOR-MAPPED-ADDRESS` din server-reflexiva kandidat.

Peer Reflexive En Peer Reflexive-kandidat är när du får en inkommande begäran från en adress som du inte känner till. Eftersom ICE är ett autentiserat protokoll vet du att trafiken är giltig. Detta betyder bara att den andra klienten försöker kommunicera med dig från en extern adress den ännu inte känner till.

Detta händer ofta när en "klientkandidat" kommunicerar med en "serverreflexiv kandidat". En ny "NAT-mappning" skapades eftersom du kommunicerar utanför ditt subnät. Kommer du ihåg att vi sa att anslutningskontrollerna faktiskt är STUN-paket? Formatet för STUN-svar tillåter naturligtvis en peer att rapportera tillbaka den peer-reflexiva adressen.

Relä (Relay) En reläkandidat genereras med hjälp av en TURN-server.

Efter den första handskakningen med TURN-servern får du en `RELAYED-ADDRESS`, detta är din reläkandidat.

Anslutningskontroller

Vi känner nu till den andra klientens "användarfragment", "lösenord" och kandidater. Vi kan nu försöka ansluta! Alla kandidater matchas med varandra, så om du har 3 kandidater på varje sida har du nu 9 kandidatpar.

Visuellt ser det ut så här:

Kandidatval

Den kontrollerande och kontrollerade agenten skickar a trafik över varje par. Detta behövs om en agent står bakom en "adressberoende mappning" (`Address Dependent Mapping`). Detta kommer att skapa en ny "peer reflexive kandidat".

Varje "kandidatpar" som lyckades skicka nätverkstrafik befordras till ett "giltigt kandidatpar". Den kontrollerande agenten nominerar sedan ett giltigt kandidatpart. Detta blir det "nominerade paret". Den kontrollerande och kontrollerade agenten försöker sedan ytterligare en runda dubbelriktad kommunikation. Om det lyckas blir det "nominerade paret" det "utvalda kandidatparet" (`Selected Candidate Pair`)! Detta par används sedan under resten av sessionen.

Startar om

Om ditt `Selected Candidate Pair` slutar fungera av någon anledning (NAT-mappning upphör att gälla, TURN-server kraschar) kommer ICE-agenten att få status `Failed`. Båda agenterna kan startas om och kommer att göra hela processen om igen.

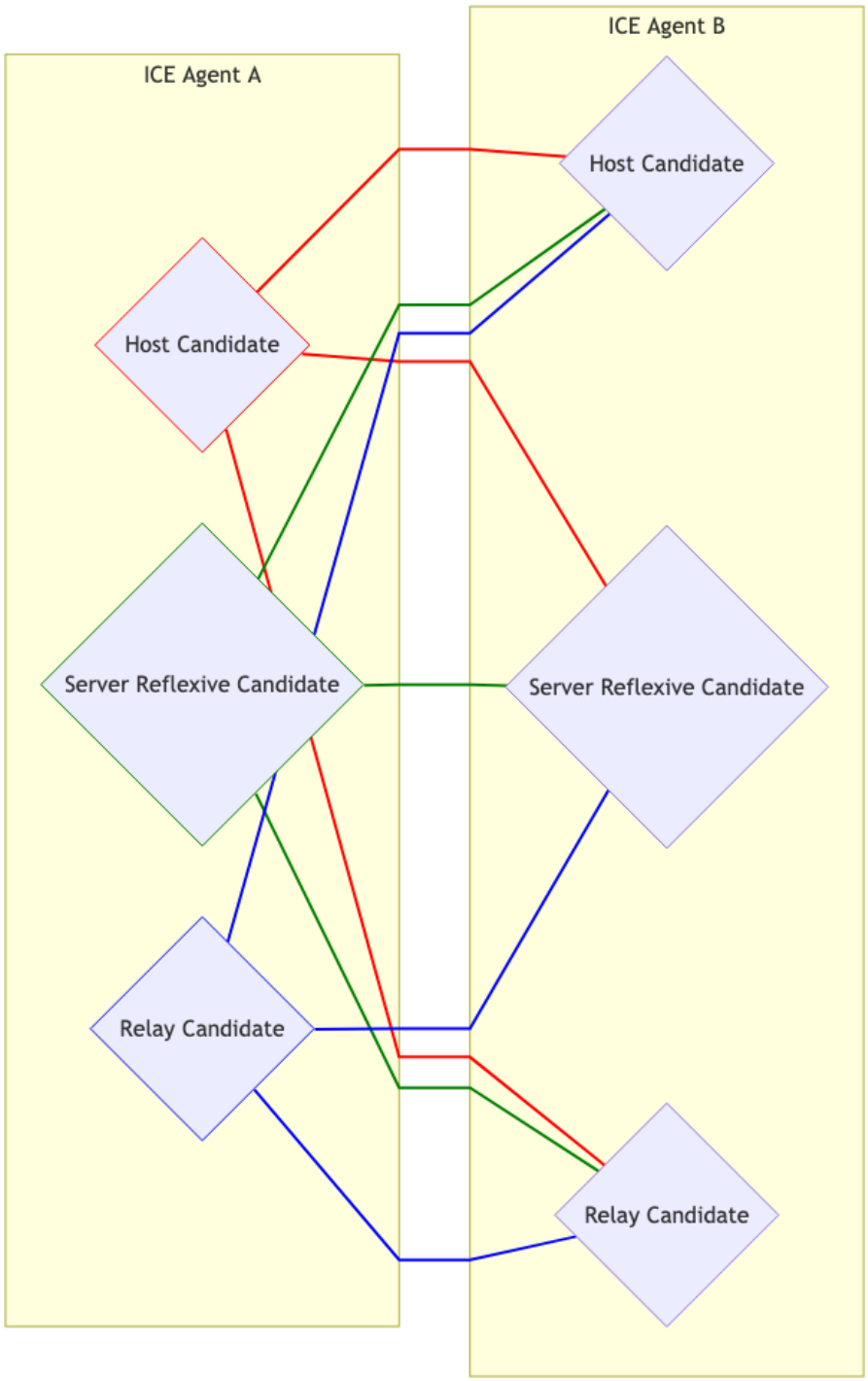


Figure 6: Anslutningskontroller

Säkerhet och WebRTC

Varje WebRTC-anslutning är autentiserad och krypterad. Du kan vara säker på att en tredje part inte kan se vad du skickar eller infoga falska meddelanden. Du kan också vara säker på att WebRTC-agenten som genererade sessionsbeskrivningen är den du kommunicerar med.

Det är mycket viktigt att ingen kan modifiera dina meddelanden. Det är ok om en tredje part läser sessionsbeskrivningen under transport. WebRTC har dock inget skydd mot att det ändras. En angripare kan utföra en man-i-mitten-attack genom att ändra ICE-kandidaterna och uppdatera certifikatets fingeravtryck.

Hur fungerar det?

WebRTC använder två befintliga protokoll, Datagram Transport Layer Security (DTLS) och Secure Real-time Transport Protocol (SRTP).

Med DTLS kan du sätta upp en session och sedan utbyta data säkert mellan två parter. Det är ett syskon till TLS, samma teknik som används för HTTPS, men DTLS använder UDP istället för TCP som transportlager. Det betyder att protokollet måste hantera tappade paket. SRTP är speciellt utformat för säkert skicka ljud och bild. Det finns några optimeringar vi kan göra genom att använda det protokollet istället för DTLS.

DTLS används först. Det gör en handskakning över anslutningen från ICE. DTLS är ett klient/serverprotokoll, så en sida måste starta handskakningen. Klient/serverroller väljs under uppsättningen. Under DTLS-handskakningen erbjuder båda sidor ett certifikat. När handskakningen är klar jämförs detta certifikat med certifikat-hashen i sessionsbeskrivningen. Detta för att säkerställa att handskakningen gjorts med den WebRTC-agent du förväntade dig. DTLS-anslutningen kan sen användas för vanlig DataChannel-kommunikation.

För att skapa en SRTP-session initialiserar vi den med hjälp av nycklarna som genereras med DTLS. SRTP har ingen handskakningsmekanism, så vi måste använda nycklar som genererats på något annat sätt. När detta är gjort kan vi skicka media krypterat över SRTP!

Grundläggande säkerhet

För att förstå tekniken som presenteras i detta kapitel finns det några termer som är bra att känna till. Kryptografi är ett knepigt ämne så det kan vara värt att konsultera andra källor också.

Klartext och Kryptotext Klartext är originaltexten man skickar in till ett chiffer. Kryptotext är resultatet av en chiffer.

Kryptering Ett chiffer är en serie steg som tar vanlig text till kryptotext. Chiffret kan sedan vändas så att du kan ta din kryptotext tillbaka till klartext.

Ett chiffer har vanligtvis en nyckel för att ändra sitt beteende. En annan term för detta är kryptering och dekryptering.

Ett enkelt chiffer är ROT13. Varje bokstav flyttas 13 tecken framåt. För att dekryptera chiffret flyttar du 13 tecken bakåt. Den klara texten `HEJSAN` skulle bli krypteringstexten `URWFNA`. I det här fallet är krypteringen ROT och nyckeln är 13.

Hashfunktioner Hash är en enkelriktad funktion som genererar en kontrollsumma. Från samma meddelande genererar den samma kontrollsumma varje gång. Det är viktigt att funktionen *inte* är reversibel. Om du har en kontrollsumma ska du inte kunna lista ut meddelandet som den genererades från. Hashing är användbart när du vill bekräfta att ett meddelande inte har manipulerats.

En enkel hash funktion skulle vara att bara ta varannan bokstav. `HEJSAN` skulle bli `HJA`. Du kan inte anta att `HEJSAN` var ingången, men du kan bekräfta att `HEJSAN` skulle vara en matcha kontrollsumman.

Kryptografi för offentlig / privat nyckel Public/Private Key Cryptography beskriver vilken typ av kod som DTLS och SRTP använder. I det här systemet har du två nycklar, en offentlig och en privat nyckel. Den offentliga nyckeln är för kryptering av meddelanden och är säker att dela. Den privata nyckeln är för dekryptering och ska aldrig delas. Det är den enda nyckeln som kan dekryptera meddelanden som är krypterade med den offentliga nyckeln.

Diffie–Hellmans nyckelöverföring Diffie–Hellman nyckelöverföring tillåter två användare som aldrig har träffats tidigare att skapa en delad hemlighet säkert över internet. Användaren `A` kan skicka en hemlighet till användaren `B` utan att oroa sig för avlyssning. Detta beror på svårigheten att bryta det diskreta logaritmproblemet. Du behöver inte helt förstå hur det fungerar, men det är bra att veta att det här är tekniken DTLS-handskakningen är baserad på.

Wikipedia har ett exempel på hur det fungerar här.

Pseudorandom-funktioner En Pseudorandom-funktion (PRF) är en fördefinierad funktion för att generera ett värde som verkar slumpmässigt. Det kan ta flera indata och generera en enda utdata.

Key Derivation Function Key Derivation är en typ av pseudorandom-funktion. Key Derivation är en funktion som används för att göra en nyckel starkare. Ett vanligt sätt är nyckelsträckning (key stretching).

Låt oss säga att du får en nyckel som är 8 byte. Du kan använda en KDF för att göra den starkare.

Nonce En nonce är ytterligare indata till ett chiffer. Detta är så att du kan få olika utdata från krypteringen, även om du krypterar samma meddelande flera gånger.

Om du krypterar samma meddelande tio gånger kommer krypteringen att ge dig samma kryptotext 10 gånger. Genom att använda ett nonce kan du få olika indata medan du fortfarande använder samma nyckel. Det är viktigt att du använder ett nytt nonce för varje meddelande! Annars ger det ger det inte någon extra säkerhet.

Meddelandeautentiseringskod En meddelandeautentiseringskod (Message Authentication Code) är en hash som placeras i slutet av ett meddelande. En MAC visar att meddelandet kommer från den användare du förväntade dig.

Om du inte använder en MAC kan en angripare infoga ogiltiga meddelanden. Efter dekryptering skulle du bara ha skräp eftersom de inte vet nyckeln.

Nyckelrotation Nyckelrotation är praxis att byta ut nyckeln med jämna mellanrum. Detta gör att det inte är lika allvarligt att en nyckel blivit stulen. Om en nyckel blir stulen eller har läckt är det mindre data som kan dekrypteras.

DTLS

DTLS (Datagram Transport Layer Security) tillåter två klienter att etablera säker kommunikation utan någon befintlig konfiguration. Även om någon lyssnar på konversationen kan de inte dekryptera meddelandena.

För att en DTLS-klient och en server ska kunna kommunicera måste de komma överens om ett chiffer och en nyckel. De bestämmer dessa värden genom att göra ett DTLS-handskakning. Under handskakningen är meddelandena i klartext. När en DTLS-klient och server har utbytt tillräckligt med information för att börja kryptera skickar den en `Change Cipher Spec`. Efter detta meddelande kommer varje efterföljande meddelande att vara krypterat!

Paketformat

Varje DTLS-paket börjar med en rubrik.

Innehållstyp Du kan förvänta dig följande typer (Content Type):

- `20` - Ändra krypteringsspecifikation (Change Cipher Spec)
- `22` - Handskakning (Handshake)
- `23` - Applikationsdata (Application Data)

`Handskakning` används för att utbyta detaljer för att starta sessionen. `Ändra krypteringsspecifikation` används för att meddela den andra sidan att allt kommer att krypteras. `Applikationsdata` är de krypterade meddelandena.

Version Version kan antingen vara `0x0000feff` (DTLS v1.0) eller `0x0000fefd` (DTLS v1.2) det finns ingen v1.1.

Epoch Epoken börjar vid `0` men blir `1` efter en `Ändra krypteringsspecifikation`. Alla meddelanden med en epok som inte är noll är krypterade.

Sekvensnummer Sekvensnummer används för att hålla meddelanden i ordning. Varje meddelande ökar sekvensnumret. När epoken ökas börjar sekvensnumret om.

Längd och data Datan (Payload) som skickas är specifik för varje "innehållstyp" (`Content Type`). För en `Applikationsdata` är det krypterad data. För `Handskakning` kommer det att vara olika beroende på meddelandet. Längden säger bara hur stor datan är.

Handskakningens tillstånd

Under handskakningen utbyter klienten och servern en serie meddelanden. Dessa meddelanden grupperas i stegar. Varje stege kan ha flera meddelanden (eller bara ett). En stege är inte klar förrän alla meddelanden i stegen har tagits emot. Vi kommer att beskriva syftet med varje meddelande mer detaljerat nedan.

ClientHello ClientHello är det första meddelandet som skickas av klienten. Den innehåller en lista med attribut. Dessa attribut berättar för servern vilka chiffer och funktioner som klienten stöder. För WebRTC så väljer vi också SRTP-chiffer med ClientHello. Den innehåller också slumpmässig data som kommer att användas för att generera nycklarna för sessionen.

HelloVerifyRequest HelloVerifyRequest skickas av servern till klienten. Det är för att se till att klienten avsåg att skicka begäran. Klienten skickar sedan ClientHello igen, men med ett token den fick från HelloVerifyRequest:et.

ServerHello ServerHello är svaret från servern för konfigurationen av sessionen. Den innehåller vilket chiffer som ska användas när den här sessionen är klar. Den innehåller också serverns slumpmässiga data.

Certifikat Certifikatet (Certificate) innehåller certifikatet för klienten eller servern. Detta används för att unikt identifiera vem vi kommunicerade med. Efter att handskakningen är över ser vi till att detta certifikat när det hashas matchar fingeravtrycket i `SessionDescription`.

ServerKeyExchange/ClientKeyExchange Dessa meddelanden används för att överföra den offentliga nyckeln. Vid start genererar klienten och servern båda ett knappsats. Efter handskakningen kommer dessa värden att användas för att generera en `Pre-Master Secret`.

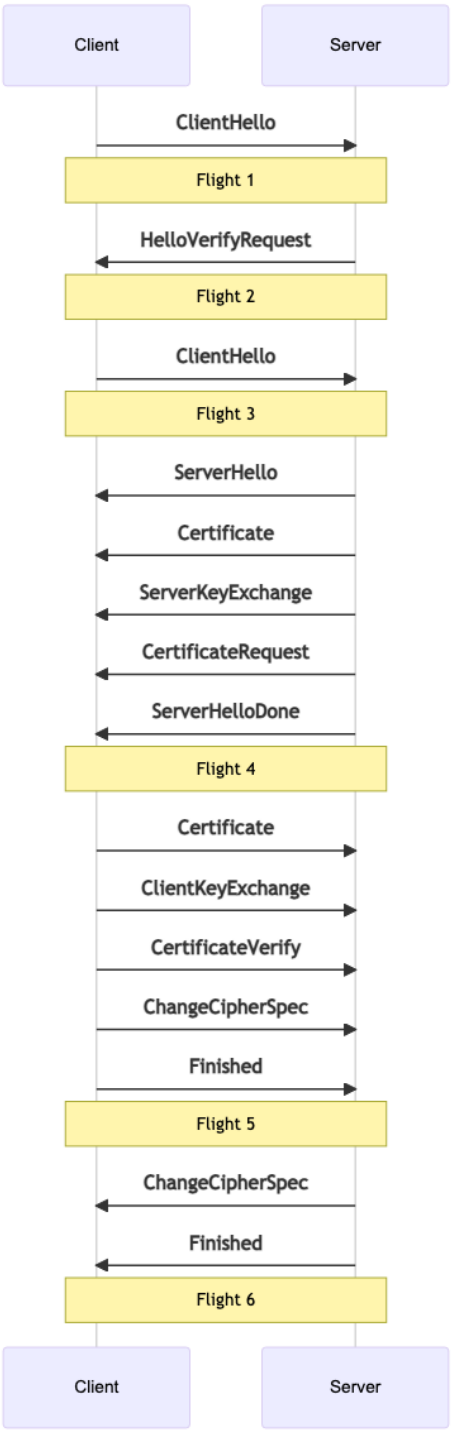


Figure 7: Handskakning

CertificateRequest En CertificateRequest skickas av servern som meddelar klienten att den vill ha ett certifikat. Servern kan antingen begära eller kräva ett certifikat.

ServerHelloDone ServerHelloDone meddelar klienten att servern är klar med handskakningen.

CertificateVerify CertificateVerify är hur avsändaren visar att den har den privata nyckeln som skickas i certifikatmeddelandet.

ChangeCipherSpec ChangeCipherSpec informerar mottagaren om att allt som skickas efter detta meddelande kommer att krypteras.

Färdiga Färdig (Finished) är krypterat och innehåller en hash av alla skickade meddelanden. Det används för att verifiera att handskakningen inte har blivit manipulerad.

Nyckelgenerering

När handskakningen är klar kan du börja skicka krypterad data. Chiffret valdes av servern och finns i ServerHello. Men hur valdes nyckeln?

Först genererar vi en `Pre-Master Secret`. För att göra det värde används Diffie–Hellman på nycklarna som byts via `ServerKeyExchange` och `ClientKeyExchange`. Detaljerna varierar beroende på vilket chiffer vi valt.

Därefter genereras en `Master Secret`. Varje version av DTLS har en egen pseudorandom-funktion. För DTLS 1.2 tar funktionen `Pre-Master Secret` och slumpvärdena i `ClientHello` och `ServerHello`. Resultatet från att köra pseudorandom-funktionen är vår `Master Secret` som används för chiffret.

Utbyta applikationsdata

Arbetshästen för DTLS är `ApplicationData`. Nu när vi har ett initialiserat chiffer kan vi börja kryptera och skicka data.

Meddelanden med `applikationsdata` använder den DTLS-header vi beskrev tidigare. Datan är skickas som krypterad text. Du har nu en fungerande DTLS-session och kan kommunicera säkert.

DTLS har många fler intressanta funktioner som till exempel omförhandling (renegotiation). Men eftersom de inte används i WebRTC hoppar vi över det.

SRTP

SRTP är ett protokoll utformat speciellt för kryptering av RTP-paket. För att starta en SRTP-session anger du dina nycklar och chiffer. Till skillnad från

DTLS har den ingen handskakningsmekanism. All konfiguration och nycklar genererades under DTLS-handskakningen.

DTLS tillhandahåller ett dedikerat API för att exportera nycklarna som ska användas av en annan process. Detta definieras i RFC 5705.

Sessionsskapande

SRTP definierar en Key Derivation-funktion som används på indatat. När du skapar en SRTP-session skickas indatat genom funktionen för att generera nycklar för vår SRTP-kryptering. Efter detta kan du gå vidare till att skicka ljud och bild.

Utbyta media

Varje RTP-paket har ett 16-bitars sekvensnummer. Dessa sekvensnummer används för att hålla ordning på paket, som en primär nyckel. Under ett samtal kommer numret bli för stort och "rulla över". SRTP håller reda på det och kallar detta för övergångsräknare (rollover counter).

Vid kryptering av ett paket använder SRTP övergångsräknaren och sekvensnumret som ett nonce. Detta för att säkerställa att även om du skickar samma data två gånger kommer krypteringstexten att vara annorlunda. Detta är viktigt för att förhindra att en angripare identifierar mönster eller försöker en uppspelningsattack (replay attack).

Varför är nätverket så viktigt i realtidskommunikation?

Nätverk är den begränsande faktorn i realtidskommunikation. I en idealisk värld skulle vi ha obegränsad bandbredd och paket skulle komma omedelbart. Detta är dock inte fallet. Nätverk är begränsade och villkoren kan ändras när som helst. Att mäta och observera nätverksförhållanden är också ett svårt problem. Du kan få olika beteenden beroende på hårdvara, programvara och konfigurationen av den.

Realtidskommunikation medför också ett problem som inte finns i många andra domäner. För en webbutvecklare är det inte avgörande om din webbplats är långsammare i vissa nätverk. Så länge all data kommer fram är användarna nöjda. Med WebRTC, om din data kommer fram sent är den värdelös. Ingen bryr sig om vad som sagts i ett konferenssamtal för 5 sekunder sedan. Det betyder att när man utvecklar ett realtidskommunikationssystem måste man göra en avvägning. Vad är min tidsgräns och hur mycket data kan jag skicka?

Detta kapitel handlar om de koncept som gäller både data- och mediekommunikation. I senare kapitel går vi igenom mer i detalj hur WebRTCs media- och data-subsystem löser dessa problem.

Vilka egenskaper har nätverk som gör det svårt?

Kod som effektivt fungerar i alla nätverk är komplicerad. Du har många olika faktorer, och de kan alla påverka varandra på subtila sätt. Här är några av de vanligaste problemen som utvecklare stöter på.

Bandbredd Bandbredd är den maximala datahastigheten som kan uppnås via en viss väg genom nätet. Det är viktigt att komma ihåg att värdet kommer att ändras under tiden. Bandbredden ändras längs en rutt (route) beroende på om fler (eller färre) människor använder den.

Sändningstid och rundturstid (Round Trip Time) Sändningstid är hur lång tid det tar för ett paket att anlända till sitt mål. Precis som bandbredd är den inte konstant utan kan variera när som helst.

```
transmission_time = mottagningstid - send_time
```

För att beräkna sändningstiden behöver klockorna på avsändaren och mottagaren vara synkroniserade med millisekunders precision. Till och med en väldigt liten avvikelse ger oss en opålitlig mätning av sändningstiden. Eftersom WebRTC används i väldigt blandade miljöer är det nästan omöjligt att förlita sig på perfekt tidssynkronisering mellan klienterna.

Tidsmätning tur och retur är en bättre lösning när klocksynkronisering inte är möjligt.

Istället för att använda distribuerade klockor skickar en WebRTC-klient ett specialpaket med sin egen tidsstämpel `sendertime1`. En mottagande klient tar emot paketet och skickar tillbaka tidsstämpeln till avsändaren. När den ursprungliga avsändaren får tillbaka tidsstämpeln drar den av `sendertime1` från aktuella tiden `sendertime2`. Denna tidsdelta kallas “round-trip propagation delay” eller rundturstid.

```
rtt = sendertime2 - sendertime1
```

Hälften av tur- och returresan anses vara en tillräckligt bra approximation av sändningstiden. Denna lösning är dock inte utan nackdelar. Den antar att det tar lika lång tid att skicka som att ta emot paket. Men i mobilnätverk kan sändnings- och mottagnings-tider vara väldigt olika. Du har kanske märkt att uppladdningshastigheter på din telefon nästan alltid är lägre än nedladdningshastigheter.

```
transmission_time = rtt/2
```

Tekniken WebRTC använder för att mäta rundturstid beskrivs mer detaljerat i kapitlet om RTCP Sender and Receiver Reports.

Jitter Jitter är det faktum att “sändningstid” (`Transmission Time`) kan variera för varje paket. Dina paket kan bli fördröjda, men sen kommer alla på en gång.

Paketförlust Paketförlust är när meddelanden går förlorade vid överföring. Förlusten kan vara jämn, eller så kan den komma i korta perioder. Detta kan bero på olika nätverkstyper som satellit eller Wi-Fi, eller introduceras av programvara längs vägen.

Maximal överföringsenhet Maximal paket storlek (MTU) är gränsen för hur stort ett enskilt data-paket kan vara. Nätverk tillåter dig inte att skicka hur stora meddelanden som helst. På protokollnivå kan meddelanden behöva delas upp i flera mindre paket.

MTU kommer också att skilja sig beroende på vilken nätverksväg du tar. Du kan använda ett protokoll som Path MTU Discovery för att räkna ut den största paketstorleken du kan skicka utan att det måste fragmenteras.

Trängsel

Trängsel är när nätverksgränserna har uppnåtts. Detta beror vanligtvis på att du har nått den maximala bandbredd som den aktuella rutten klarar. Eller så kan det vara operatören som begränsar din datamängd per timme.

Trängsel visar sig på många olika sätt. Det finns inget standardiserat beteende. I de flesta fall brukar nätverket tappa överflödiga paket. I andra fall kommer nätverket att buffra. Detta gör att sändningstiden för dina paket kommer att öka. Du kan också märka mer jitter när ditt nätverk blir överbelastat. Detta här är ett område där mycket ändras och nya algoritmer för att upptäcka överbelastning utvecklas fortfarande.

Dynamisk

Nätverk är oerhört dynamiska och förhållandena kan förändras snabbt. Under ett samtal kan du skicka och ta emot hundratusentals paket. Dessa paket kommer att resa genom flera nätverkshopp. De här nätverkshoppen kommer att delas med miljontals andra användare. Även i ditt lokala nätverk kan du ha HD-filmer som laddas ner eller kanske en enhet bestämmer sig för att ladda ner en stor programuppdatering.

För att ha ett bra videosamtal räcker det inte att bara mäta nätverket vid starten. Du måste ständigt utvärdera och hantera alla förändringar som kommer från nätverket och programvaran som är inblandad.

Lösa paketförlust

Att hantera paketförlust är det första problemet att lösa. Det finns flera sätt att lösa det, alla med olika för- och nackdelar. Dels beror det på vad du skickar och hur mycket latens som är acceptabelt. Det är också viktigt att notera att inte all paketförlust är kritisk. Att förlora lite video behöver inte vara ett problem, det mänskliga ögat kanske inte även uppfattar det. Att förlora en användares textmeddelanden är ödesdigert.

Låt oss säga att du skickar 10 paket, och paket 5 och 6 går förlorade. Här är några exempel på hur man kan lösa det.

Bekräftelser

Bekräftelser är när mottagaren meddelar avsändaren om varje paket de har fått. Avsändaren är medveten om paketförlust när den får ett bekräftelse för samma paket två gånger som inte är det sista paketet. När avsändaren får ett `ACK` för paket nummer 4 två gånger, vet den att paket nummer 5 inte har levererats än.

Selektiva bekräftelser

Selektiva bekräftelser är en förbättring jämfört med bekräftelser. En mottagare kan skicka ett `SACK` som bekräftar flera paket och meddelar avsändaren om luckor. Om avsändaren får ett `SACK` för paket 4 och 7. Då vet den att den måste skicka paket 5 och 6 igen.

Negativa bekräftelser

Negativa bekräftelser löser problemet tvärtom. Istället för att meddela avsändaren vad den har fått, meddelar mottagaren avsändaren vad den saknar. I vårt fall kommer ett `NACK` att skickas för paket 5 och 6. Avsändaren får bara reda på paket som mottagaren vill ska skickas igen.

Forward Error Correction

Forward Error Correction är ett sätt att tåla viss paketförlust. Avsändaren skickar redundant data, vilket innebär att ett enstaka förlorat paket inte påverkar den slutliga strömmen. En populär algoritm för detta är Reed–Solomon-felkorrigering.

Detta minskar latensen och komplexiteten för att skicka och hantera bekräftelser. Forward Error Correction är ett slöseri med bandbredd om nätverket du använder inte har någon paketförlust.

Hantera jitter

Jitter finns i de flesta nätverk. Även på ett lokalt nätverk har du många enheter som skickar data med varierande hastigheter. Du kan enkelt se jitter genom att pinga en annan enhet med kommandot `ping` och notera variationen i tiden det tar att få svar.

För att hantera jitter använder du en JitterBuffer. En JitterBuffer garanterar en stabil leveranstid för paketen. Nackdelen är att JitterBuffer lägger till lite extra latens för varje paket som kommer i tid. Fördelen är att sena paket inte orsakar något jitter. Föreställ dig att du ser följande ankomsttider för några paket under ett samtal:

```
* time=1.46 ms
```

```
* time=1.93 ms
* time=1.57 ms
* time=1.55 ms
* time=1.54 ms
* time=1.72 ms
* time=1.45 ms
* time=1.73 ms
* time=1.80 ms
```

I det här fallet skulle cirka 1,8 ms vara ett bra val. Paket som kommer försent kommer att använda vårt latensfönster. Paket som kommer tidigt kommer att fördröjas lite och kan fylla ut luckan från sena paket. Det betyder att videoströmmen inte längre stammar och ger klienten en jämn leveranshastighet.

JitterBuffer-funktion

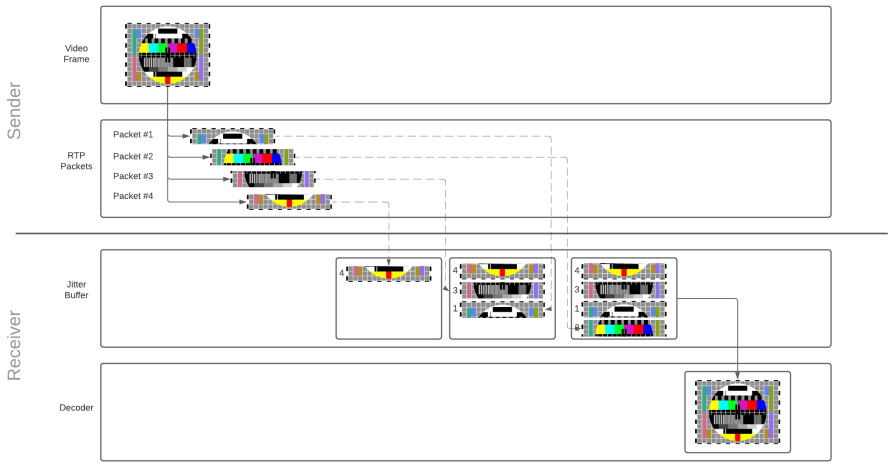


Figure 8: JitterBuffer

Varje paket läggs till jitterbufferten så snart det tas emot. När det finns tillräckligt med paket för att återskapa hela bilden släpps paketen som utgör bilden från bufferten och skickas för avkodning. Avkodaren i sin tur ritar upp bilden på användarens skärm. Eftersom jitterbufferten har en begränsad storlek kommer paket som fastnat för länge i bufferten att kasseras.

Läs mer om hur video konverteras till RTP-paket och varför rekonstruktion är nödvändig i kapitlet om mediekommunikation.

`jitterBufferDelay` visar tydligt ditt nätverks prestanda och hur den påverkar videouppspelningens jämnhet. Det är en del av WebRTC Statistics API och spelar roll för mottagarens inkommande videoström. Förseningen definierar hur

mycket tid bilder spenderar i jitterbuffern innan de skickas vidare för avkodning. En lång jitterbuffertfördröjning innebär att ditt nätverk är överbelastat.

Upptäcka trängsel

Innan vi ens kan lösa trängsel måste vi upptäcka det. För att upptäcka det använder vi en trängselregulator (congestion controller). Detta är ett komplicerat ämne och förändras fortfarande snabbt. Nya algoritmer publiceras fortfarande och testas. På en hög nivå fungerar de alla på samma sätt. En överbelastningsregulator ger en uppskattning av bandbredd givet en viss indata. Det här är några möjliga indata:

- **Paketförlust** - Paket tappas när nätverket blir överbelastat.
- **Jitter** - När nätverksutrustningen blir mer överbelastad kommer köande av paket orsaka oregelbundna leveranstider.
- **Rundturstid** - Paket tar längre tid att anlända när uppkopplingen är överbelastad. Till skillnad från jitter fortsätter rundturstiden bara att öka.
- **Explicit trängselnotis** - Nyare nätverk kan märka paket som riskerar att tappas för att undvika trängsel.

Dessa värden måste mätas kontinuerligt under samtalet. Användningen av nätverket kan öka eller minska, så den tillgängliga bandbredden kan ständigt förändras.

Lösa trängsel

Nu när vi har en uppskattad bandbredd måste vi justera vad vi skickar. Hur vi justerar beror på vilken typ av data vi vill skicka.

Skicka långsammare

Att begränsa hastigheten med vilken du skickar data är ett enkelt sätt att förhindra trängsel. Trängselregulatorn ger dig en uppskattning, och det är avsändarens ansvar att bedöma hur mycket hastigheten ska begränsas.

Detta är den vanligaste metoden som brukar användas. Med protokoll som TCP görs allt detta av operativsystemet och helt transparent för både användare och utvecklare.

Skicka mindre

I vissa fall kan vi skicka mindre information för att undvika att slå i taket. Vi kan också ha hårda tidsbegränsningar för när våra data ska levereras, så vi kanske kan inte skicka långsammare. Det här är en typisk begränsning som realtidsmedier faller under.

Om vi inte har tillräckligt med bandbredd kan vi sänka kvaliteten på den video vi skickar. Detta kräver en tät feedbackslinga mellan din videoencoder och din trängselregulator.

Vad får jag från WebRTCs mediakommunikation?

WebRTC låter dig skicka och ta emot ett obegränsat antal ljud- och videoströmmar. Du kan lägga till och plocka bort strömmar när som helst under ett samtal. Dessa strömmar kan alla vara oberoende, eller de kan buntas ihop! Du kan skicka ett videoflöde som visar din presentation och samtidigt inkludera ljud och video från din webbkamera.

WebRTC-protokollet är kodek-agnostiskt. Den underliggande transporten stöder allt, även kodeks som inte finns ännu! Det är dock fullt möjligt att WebRTC-agenten som du kommunicerar med inte har det nödvändiga stödet för att kunna acceptera den.

WebRTC är också utformat för att hantera dynamiska nätverksförhållanden. Under ett samtal kan din bandbredd öka eller minska. Kanske upplever du plötsligt mycket paketförlust. Protokollet är utformat för att hantera allt detta. WebRTC svarar på nätverksförhållanden och försöker ge dig bästa möjliga upplevelse utifrån de tillgängliga resurserna.

Hur fungerar det?

WebRTC använder två befintliga protokoll RTP och RTCP, båda definierade i RFC 1889.

RTP (Real-time Transport Protocol) är protokollet som media skickas över. Det var utformat för att möjliggöra leverans av video i realtid. Det innehåller inga regler kring latens eller tillförlitlighet, men ger dig verktygen för att implementera dem. RTP ger dig strömmar så att du kan köra flera medieflöden över en anslutning. Det ger dig också tidpunkten och beställningsinformationen du behöver för att mata en mediapipeline.

RTCP (RTP Control Protocol) är det protokoll som skickar metadata om samtalet. Formatet är mycket flexibelt och låter dig lägga till all möjlig metadata du vill ha. Detta används till exempel för att kommunicera statistik om samtalet. Det används också för att hantera paketförluster och för överbelastningskontroll. Det ger dig den dubbelriktade kommunikation som krävs för att du ska kunna anpassa dig till varierande nätverksförhållanden.

Latens mot kvalitet

Realtidsmedia handlar om att göra avvägningar mellan latens och kvalitet. Ju mer latens du är villig att tolerera, desto högre kvalitet kan du förvänta dig.

Begränsningar i verkliga världen

Dessa begränsningar orsakas alla av den verkliga världens begränsningar. De är alla egenskaper hos ditt nätverk som du kommer att behöva hantera.

Video är komplext

Att transportera video är inte lätt. För att lagra 30 minuter okomprimerad 720 8-bitars video behöver du circa 110 GB. Med dessa siffror kommer ett konferenssamtal på 4 personer inte gå att genomföra. Vi behöver ett sätt att göra det mindre, och svaret är videokomprimering. Det kommer dock att medföra andra nackdelar.

Video 101

Vi kommer inte att gå igenom videokomprimering alltför detaljerat, bara tillräckligt för att förstå varför RTP är utformat som det är. Videokomprimering kodar video till ett nytt format som kräver färre bitar för att representera samma video.

Förlustfri och inexakt komprimering

Du kan koda video för att vara förlustfri (ingen information går förlorad) eller inexakt (information kan gå förlorad). Eftersom förlustfri kodning kräver att mer data skickas till mottagaren, vilket ger högre latens och fler tappade paket, använder RTP vanligtvis inexakt komprimering även om videokvaliteten inte blir lika bra.

Inexakt videokomprimering

Videokomprimering finns i två typer. Den första är enkel komprimering vilket betyder att man minskar bitarna som används för att beskriva en bild en bild i taget. Samma teknik används för att komprimera stillbilder, som till exempel JPEG-komprimering.

Den andra typen är dubbelriktad komprimering. Eftersom video består av många bilder letar vi efter sätt att inte skicka samma information två gånger.

Dubbelriktad bildruta

Du har då tre bildtyper:

- **I-Bild** - En komplett bild, kan avkodas utan något annat
- **P-Bild** - En komprimerad bild som bara innehåller det som skiljer den från föregående bild.
- **B-Bild** - En dubbelriktad bildruta. Bilden återställs med hjälp av information från både den föregående och den efterföljande bildrutan.

Följande är visualisering av de tre bildtyperna.

Video är känslig

Videokomprimering är otroligt tillståndsbaserad, vilket gör den svår att överföra via internet. Vad händer om du förlorar en del av en I-Bild? Hur vet en P-Bild

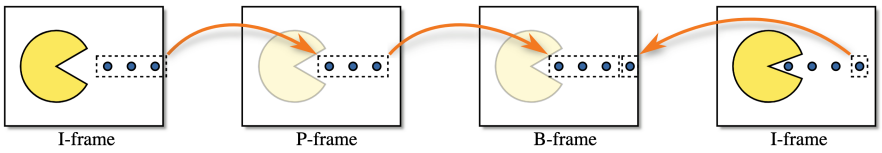


Figure 9: Frame types

vad den ska ändra? Eftersom videokomprimering blivit mer och mer komplex har detta blivit ett ännu större problem. Lyckligtvis har RTP och RTCP en lösning på problemet.

RTP

Paketformat

Varje RTP-paket har följande struktur:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       Sequence Number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Synchronization Source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            Contributing Source (CSRC) identifiers             |
|                               ....                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Payload                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Version (V) `Version` är alltid 2

Padding (P) `Padding` är en boolean som kontrollerar om datan har blivit utfylld till ett jämnt värde.

Den sista byten av data visar hur många utfyllnadsbytes som lagts till.

Tillägg (X) Om inställt kommer RTP-rubriken att ha tillägg. Detta beskrivs närmare nedan.

CSRC-antal (CC) Mängden `CSRC`-identifierare som följer efter `SSRC` och före datan.

Markör (M) Markörbiten har ingen fördefinierad betydelse och kan användas som man vill.

I vissa fall sätts den när en användare pratar. Den används också ofta för att markera en nyckelbild (`key frame`).

Payload Type (PT) `Payload Type` är en unik identifierare som beskriver vilken codec som används för detta paket.

För WebRTC är `Payload Type` dynamisk. VP8 i ett samtal kan vara annorlunda än ett annat. Avsändaren i samtalet bestämmer mappningen av `Payload Type` till kodekarna i sessionsbeskrivningen (`Session Description`).

Sekvensnummer Sekvensnummer används för att hålla koll på ordningen av paketen i en ström. Varje gång ett paket skickas ökas sekvensnumret med ett.

RTP är utformat för att vara användbart över nätverk där man tappar paket. Sekvensnummer ger mottagaren ett sätt att upptäcka när paket har gått förlorade.

Tidsstämpel Samplingsögonblicket för detta paket. Det här är inte en global klocka, utan hur mycket tid som har gått i mediaströmmen. Flera RTP paket kan ha samma tidsstämpel om de till exempel tillhör samma bild i en videoström.

Synkroniseringskälla (SSRC) En `SSRC` är den unika identifieraren för denna ström. Detta gör att du kan köra flera mediaströmmar över en enda RTP-ström.

Bidragande källa (CSRC) En lista som visar vilka källor (`SSRC`er) som bidrog till detta paket.

Detta används ofta för att visa vem som pratar. Om du kombinerade flera ljudflöden på serversidan till en enda RTP-ström, kan använda det här fältet för att säga “Källorna A och C pratade just nu”.

Payload Den faktiska datan. Den sista byten kan innehålla hur många utfyllnadsbytes som lagts till om utfyllnadsflaggan (`P`) är satt.

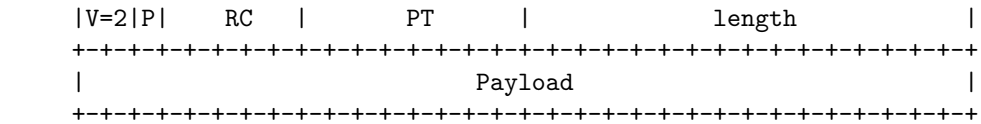
Tillägg

RTCP

Paketformat

Varje RTCP-paket har följande struktur:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



Version (V) `Version` är alltid `2`.

Padding (P) `Padding` är en boolean som kontrollerar om datan har blivit utfylld till ett jämnt värde.

Den sista byten av data visar hur många utfyllnadsbytes som lagts till.

Antal mottagningsrapporter (RC) Antalet rapporter i detta paket. Ett enda RTCP-paket kan innehålla flera händelser.

Pakettyp (PT) Unik identifierare för vilken typ av RTCP-paket detta är. En WebRTC-agent behöver inte stödja alla dessa typer, och stöd mellan klienter kan vara annorlunda. De följande är dock de som du ofta ser:

- `192` - Fullständig INTRA-frame Request (`FIR`)
- `193` - Negativa bekräftelser (`NACK`)
- `200` - Avsändarrapport
- `200` - Mottagarrapport
- `200` - Generisk RTP-feedback
- `200` - Specifik feedback för datan

Betydelsen av dessa pakettyper kommer att beskrivas mer detaljerat nedan.

Full INTRA-frame Request (FIR) och Picture Loss Indication (PLI)

Både `FIR` och `PLI`-meddelanden tjänar ett liknande syfte. Dessa meddelanden begär en fullständig nyckelbild (`I-Bild`) från avsändaren. `PLI` används när partiella bilder anländer till avkodaren som den inte kan avkoda. Detta kan hända om du hade mycket paketförlust, eller kanske avkodaren kraschade.

Enligt RFC 5104 ska `FIR` inte användas när paket eller bilder går förlorade, det är `PLI`s jobb. `FIR` begär en nyckelbild av andra skäl än paketförlust - till exempel när en ny klient går med i en videokonferens. De behöver en hel nyckelram för att starta avkodning av videoströmmen, avkodaren kommer att kasta bort alla partiella bilder tills nyckelbild kommer.

Det är en bra idé för en mottagare att begära en fullständig nyckelbild direkt efter anslutning, detta minimerar fördröjningen mellan anslutning till att en bild visas på användarens skärm.

`PLI` paket är en del av dataspecifika feedback-meddelanden.

I praktiken kommer programvara som kan hantera både `PLI` och `FIR`-paket att fungera på samma sätt i båda fallen. Den kommer att skicka en signal till kodaren för att producera en ny nyckelbild.

Negativa bekräftelser

Ett `NACK` begär att en avsändare sänder igen ett enda RTP-paket. Detta orsakas vanligtvis när ett RTP-paket går förlorat, men kan också hända eftersom det är sent.

`NACK` är mycket mer bandbreddseffektivt än att begära att hela bilden skickas igen. Eftersom RTP delar upp paket i många små bitar, begär du egentligen bara en liten saknad del. Mottagaren skapar ett RTCP-meddelande med SSRC och sekvensnummer. Om avsändaren inte har detta RTP-paket tillgängligt för att skicka igen ignorerar det bara meddelandet.

Avsändar- och mottagarrapporter

Dessa rapporter används för att skicka statistik mellan agenter. Detta kommunicerar mängden paket som faktiskt mottagits och jitter.

Rapporterna kan användas för diagnostik och trängselskontroll.

Hur RTP/RTCP löser problem tillsammans

RTP och RTCP arbetar sedan tillsammans för att lösa alla problem som orsakas av nätverk. Dessa tekniker förändras fortfarande.

Vidarekorrigering av fel

Även känd som Forward Error Correction (FEC). En annan metod för att hantera paketförlust. FEC är när du skickar samma data flera gånger, utan att det ens begärs. Detta görs på RTP-nivå, eller till och med lägre nivå via kodeken.

Om paketförlusten för ett samtal är stabil get FEC en mycket lägre latens än NACK. Tur-och-returtiden för att begära och sedan skicka om ett paketet kan vara ganska stor när man använder NACK.

Adaptiv bittakt och uppskattning av bandbredd

Nätverk är oförutsägbara och opålitliga, som vi tidigare diskuterade i kapitlet om Realtidsnätverk. Bandbredds tillgänglighet kan ändras flera gånger under en session. Det är inte ovanligt att tillgänglig bandbredd förändras dramatiskt (flera storleksordningar) inom en sekund.

Huvudidén är att justera kodningshastigheten baserat på förutsagd, aktuell och framtida tillgänglig nätverksbandbredd. Detta säkerställer att video- och ljudsignaler av bästa möjliga kvalitet överförs och att anslutningen inte tappas

på grund av trängsel i nätverket. Heuristik som modellerar nätverksbeteendet och försöker förutsäga det kallas uppskattning av bandbredd.

Det finns mycket nyans i detta, så låt oss utforska det mer i detalj.

Kommunicera nätverksstatus

Det första problemet med att implementera trängselskontroll är att UDP och RTP inte kommunicerar nätverksstatus. Som avsändare har jag ingen aning om när mina paket anländer eller om de kom fram över huvud taget!

RTP/RTCP har tre olika lösningar på detta problem som alla har sina för- och nackdelar. Vad du använder beror på vilka klienter du arbetar med. Vad är topologin du arbetar med, eller till och med bara hur mycket utvecklingstid du har kvar.

Mottagarrapporter

Mottagarrapporter är RTCP-meddelanden, det ursprungliga sättet att kommunicera nätverksstatus. Du hittar dem i RFC 3550. De skickas enligt ett schema för varje SSRC och innehåller följande fält:

- **Fraktion förlorad** - Hur stor andel av paketen som har gått förlorade sedan den senaste mottagarrapporten.
- **Kumulativt antal förlorade paket** - Hur många paket som har gått förlorade under hela samtalet.
- **Utökat högsta mottagna sekvensnummer** - Vad var det senaste mottagna sekvensnumret och hur många gånger har det rullat över.
- **Interarrival Jitter** - Den rullande jittren för hela samtalet.
- **Senaste avsändarrapportens tidsstämpel** - Senast kända tid på avsändaren, används för beräkning av tur och retur-tid.

Avsändar- och mottagarrapporter (SR och RR) används tillsammans för att beräkna tur och retur-tid.

Avsändaren inkluderar sin lokala tid, `sendertime1` i avsändarrapporten (SR). När mottagaren får SR-paket skickar den tillbaka mottagarrapporten (RR). RR inkluderar bland annat `sendertime1` som just erhållits från avsändaren. Det kommer att finnas en fördröjning mellan att ta emot SR och att skicka RR. På grund av detta inkluderar RR också en "fördröjning sedan senaste avsändares rapport" - `DLSR`. `DLSR` används för att justera uppskattningen av tur och retur-tiden senare i processen. När avsändaren tar emot RR drar den bort `sendertime1` och `DLSR` från aktuell tid som vi kallar `sendertime2`. Detta tidsdelta kallas tur-och-retur-fördröjning.

```
rtt = sendertime2 - sendertime1 - DLSR
```

Rundturstid på vanlig svenska: - Jag skickar ett meddelande till dig med min klockas nuvarande tid, säg att klockan är 16:20, 42 sekunder och 420 millisekunder.

- Du skickar tillbaka samma tidsstämpel till mig. - Du inkluderar också tiden som gått från att du läste mitt meddelande tills att du skickade tillbaka meddelandet, säg 5 millisekunder. - När jag har fått tiden tillbaka kollar jag på min klocka igen. - Nu säger min klocka 16:20, 42 sekunder 690 millisekunder. - Det betyder att det tog 265 millisekunder (690 - 420 - 5) att nå till dig och tillbaka till mig. - Därför är tur-och-retur-tiden 265 millisekunder.

TMMBR, TMMBN och REMB

Nästa generation av nätverksstatusmeddelanden involverar alla mottagare som skickar meddelanden via RTCP med uttryckliga bittaktförfrågningar.

- **Tillfällig maximal medieströmförfrågan för bittakt (TMMBR)** - En mantissa/exponent för en begärd bittakt för en enda SSRC.
- **Tillfällig maximal mediaströmbittaktsmeddelande (TMMBN)** - Ett meddelande om att en förfrågan har mottagits.
- **Mottagarens beräknade maximala bittakt (REMB)** - En mantissa/exponent för en begärd bithastighet för hela sessionen.

TMMBR och TMMBN kom först och definieras i RFC 5104. REMB kom senare, det finns ett utkast i draft-alvestrand-rmcat-remb, men det blev aldrig standardiserat.

En session som använder REMB skulle se ut enligt följande:

Webbläsare använder en enkel tumregel för uppskattning av bandbredd: 1. Be omkodaren att öka bithastigheten om nuvarande paketförlust är mindre än 2%. 2. Om paketförlusten är högre än 10%, sänk bithastigheten med hälften av den aktuella paketförlustprocenten.

```
if (packetLoss < 2%) video_bitrate *= 1.08
if (packetLoss > 10%) video_bitrate *= (1 - 0.5*lossRate)
```

Denna metod fungerar bra på papper. Avsändaren tar emot uppskattning från mottagaren, ställer in kodningshastigheten till det mottagna värdet. Allt klart! Vi har anpassat oss till nätverksförhållandena.

I praktiken har REMB-metoden dock flera nackdelar.

Omkodares ineffektivitet är en av dem. När du anger en bithastighet för omkodare kommer den inte nödvändigtvis att mata ut den exakta bithastigheten du begärde. Den kan mata ut fler eller färre bitar, beroende på omkodarinställningarna och bilden som ska skickas.

Till exempel kan användning av x264-omkodaren med `tune=zerolatency` avsevärt avvika från den önskade bittakten. Här är ett möjligt scenario:

- Låt oss säga att vi börjar med att sätta bittakten till 1000 kbps.
- Omkodaren matar bara ut 700 kbps, eftersom det inte finns tillräckligt med detaljer i bilden att koda. (Också känt som “stirrar på en vägg”.)

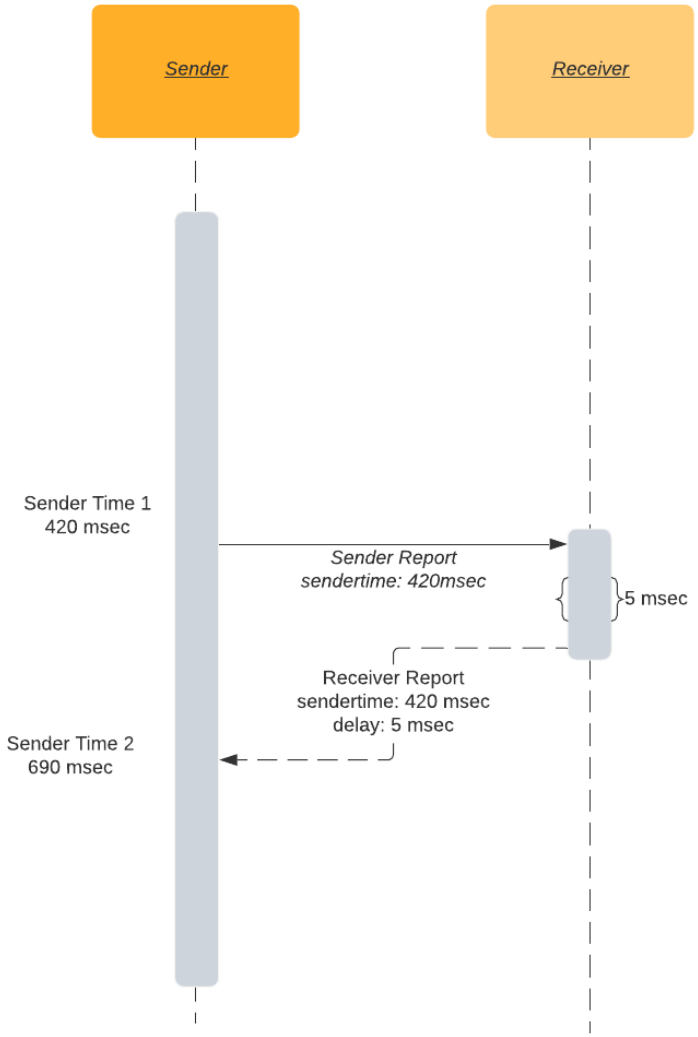


Figure 10: Rundturstid

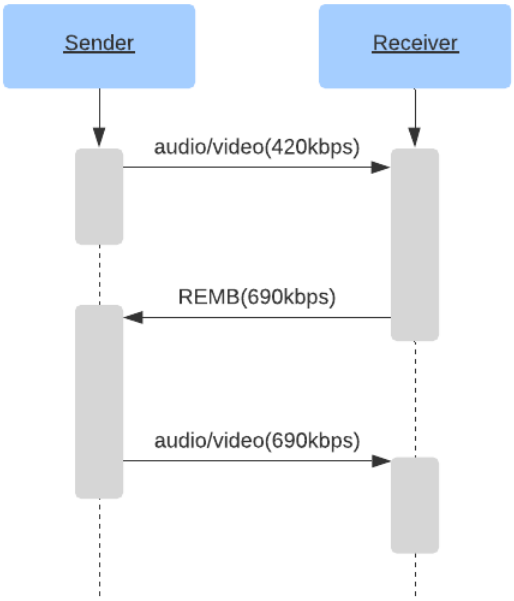


Figure 11: REMB

- Låt oss också föreställa oss att mottagaren tar emot 700 kbps-videon helt utan paketförlust. Den tillämpar då REMB-regel 1 för att öka den inkommande bithastigheten med 8%.
- Mottagaren skickar ett REMB-paket med ett förslag på 756 kbps (700 kbps * 1.08) till avsändaren.
- Avsändaren ställer in kodningshastigheten på 756 kbps.
- Kodaren matar ut en ännu lägre bithastighet.
- Denna process fortsätter att upprepa sig själv och sänker bittakten till det absoluta minimumet.

Man kan se hur detta snabbt skulle orsaka en alldeles för låg bittaktsinställning för omkodaren och på så sätt överraska användare med mycket dålig video även över en bra nätverksanslutning.

Transport Wide Congestion Control

Transport Wide Congestion Control är den senaste utvecklingen inom RTCP-nätverksstatuskommunikation.

TWCC använder en ganska enkel princip:

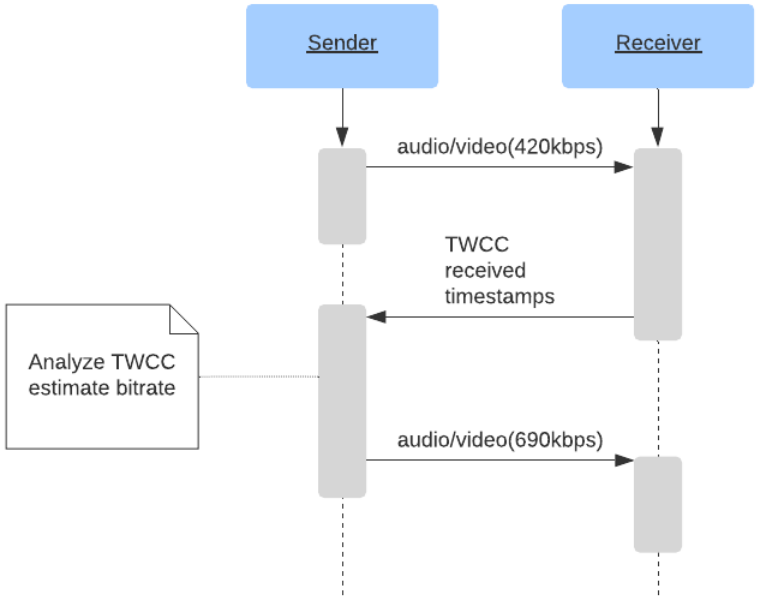


Figure 12: TWCC

Till skillnad från i REMB försöker en TWCC-mottagare inte uppskatta sin egen inkommande bithastighet. Det låter bara avsändaren veta vilka paket som togs

emot och när. Baserat på dessa rapporter har avsändaren en mycket uppdaterad uppfattning om vad som händer i nätverket.

- Avsändaren skapar ett RTP-paket med en speciell TWCC-header som innehåller en lista över paketsekvensnummer.
- Mottagaren svarar med ett speciellt RTCP-feedbackmeddelande som meddelar avsändaren om och när varje paket mottogs.

Avsändaren håller reda på skickade paket, deras sekvensnummer, storlekar och tidsstämplar. När avsändaren tar emot RTCP-meddelanden från mottagaren jämför den sändningsfördröjningarna mellan paket med mottagningsfördröjningar. Om mottagningsfördröjningarna ökar betyder det att det finns trängsel i nätverket som avsändaren måste anpassa sig till.

I diagrammet nedan är medianfördröjningen mellan paketet +20 msek, en tydlig indikator på att det är trängsel i nätverket.

! TWCC med fördröjning

TWCC tillhandahåller rådata och en utmärkt insikt i nätverksförhållanden i realtid: - Nästan omedelbar statistik över paketförluster, inte bara den procentuella förlusten utan de exakta paketen som förlorades. - Exakt skickad bittakt. - Exakt mottagningshastighet. - En jitter uppskattning. - Skillnaden i fördröjningar mellan att skicka och ta emot paket.

En trivial överbelastningskontrollalgoritm för att uppskatta den inkommande bitakten hos mottagaren från avsändaren är att summera paketstorlekar som mottagits och dela den med tiden som passerat.

Skapa en uppskattning av bandbredd

Nu när vi har information om tillståndet för nätverket kan vi göra uppskattningar om tillgänglig bandbredd. 2012 startade IETF arbetsgruppen RMCAT (RTP Media Congestion Avoidance Techniques). Denna arbetsgrupp innehåller flera inlämnade standarder för algoritmer för överbelastning. Innan dess var alla algoritmer för överbelastningskontroll egna.

Den mest använda implementeringen är “A Google Congestion Control Algorithm for Real-Time Communication” definierad i draft-alvestrand-rmcat-congestion. In kan köra i två iterationer. Först ett “förlustbaserat” pass som bara använder mottagarrapporter. Om TWCC är tillgängligt kommer det också att ta hänsyn till den informationen. Den förutspår nuvarande och framtida nätverksbandbredd genom att använda ett Kalman-filter.

Det finns också flera alternativ till GCC, till exempel NADA: A Unified Congestion Control Scheme for Real-Time Media och SCReAM - Self-Clocked Rate Adaptation for Multimedia.

Vad får jag från WebRTCs datakommunikation?

WebRTC tillhandahåller datakanaler för datakommunikation. Mellan två klienter kan du öppna 65534 datakanaler. En datakanal är UDP-baserad och alla har sina egna hållbarhetsinställningar. Som standard har varje data kanal garanterad leveransordning på skickade paket.

Om du närmar dig WebRTC från ett mediabakgrund kan datakanaler verka slösaktiga. Varför använda hela detta delsystem när du bara kan använda HTTP eller WebSockets?

Den verkliga kraften hos datakanaler är att du kan konfigurera dem så att de beter sig som UDP med oordnad/osäker leverans. Detta är nödvändigt för situationer med låg latens och höga prestanda. Du kan mäta mottrycket och försäkra dig att du bar skickar bara så mycket data som ditt nätverk klarar.

Hur fungerar det?

WebRTC använder Stream Control Transmission Protocol (SCTP), definierat i RFC 4960. SCTP är ett transportlagerprotokoll som var tänkt som ett alternativ till TCP eller UDP. För WebRTC använder vi det som ett applikationslagerprotokoll via vår DTLS-anslutning.

SCTP ger dig strömmar och varje ström kan konfigureras oberoende. WebRTC-datakanaler är bara tunna abstraktioner kring dem. Inställningarna kring hållbarhet och beställning skickas precis in i SCTP-agenten.

Datakanaler har vissa funktioner som SCTP inte kan hantera, till exempel kanaletiketter (labels). För att lösa det använder WebRTC DCEP (Data Channel Establishment Protocol) vilket definieras i RFC 8832. DCEP definierar meddelanden för att kommunicera kanaletiketten och protokollet.

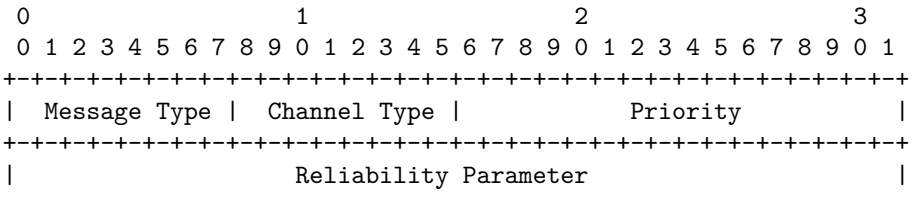
DCEP

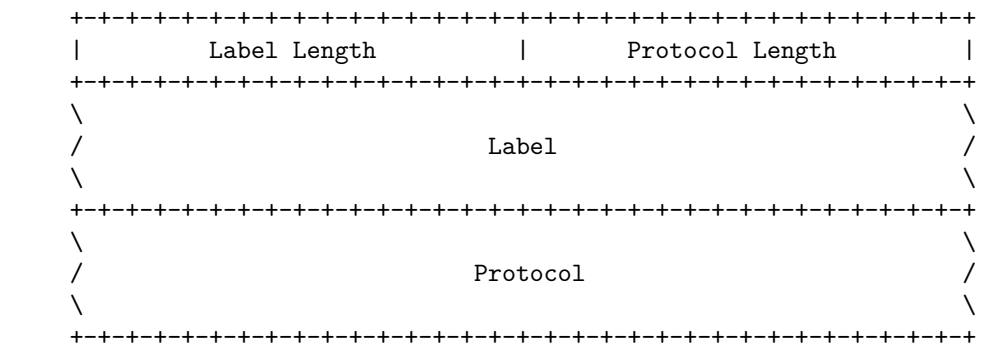
DCEP har bara två meddelanden `DATA_CHANNEL_OPEN` och `DATA_CHANNEL_ACK`. För varje datakanal som öppnas måste mottagaren svara med ett ack.

DATA_CHANNEL_OPEN

Detta meddelande skickas av WebRTC-agenten som vill öppna en kanal.

Paketformat





Message Type Message Type har alltid samma värde, `0x03`.

Channel Type Med Channel Type sätter man kanalens hållbarhet och ordningsattribut. Den kan ha följande värden:

- `DATA_CHANNEL_RELIABLE` (`0x00`) - Inga meddelanden går förlorade och de levereras i ordning.
- `DATA_CHANNEL_RELIABLE_UNORDERED` (`0x80`) - Inga meddelanden går förlorade, men kan komma i fel ordning.
- `DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT` (`0x01`) - Meddelanden kan gå förlorade efter att ha försökt ett givet antal gånger, men de kommer i ordning.
- `DATA_CHANNEL_PARTIAL_RELIABLE_REXMIT_UNORDERED` (`0x81`) - Meddelanden kan gå förlorade efter att ha försökt ett givet antal gånger och kan också komma i fel ordning.
- `DATA_CHANNEL_PARTIAL_RELIABLE_TIMED` (`0x02`) - Meddelanden kan gå förlorade om de inte kommer fram inom den önskade tiden, men de kommer fram i ordning.
- `DATA_CHANNEL_PARTIAL_RELIABLE_TIMED_UNORDERED` (`0x82`) - Meddelanden kan gå förlorade om de inte kommer fram inom önskad tid och kan komma i vilken ordning som helst.

Prioritet Datakanalens prioritet. Datakanaler med högre prioritet schemaläggs först. Stora meddelanden med lägre prioritet kommer inte att orsaka fördröjningar för meddelanden med högre prioritet.

Pålitlighetsparameter Om datakanaltypen är `DATA_CHANNEL_PARTIAL_RELIABLE` konfigurerar suffixet beteendet:

- `REXMIT` - Definierar hur många gånger avsändaren ska skicka meddelandet igen innan den ger upp.
- `TIMED` - Definierar hur länge (i ms) avsändaren ska skicka meddelandet igen innan den ger upp.

Etikett (Label) Namnet på datakanalen som en UTF-8-kodad sträng. Det kan vara en tom sträng.

Protokoll Om det är en tom sträng är protokollet ospecificerat. Om fältet innehåller text ska det vara ett protokoll registrerat i "WebSocket Subprotocol Name Registry", definierat i RFC 6455.

DATA_CHANNEL_ACK

Detta meddelande skickas av WebRTC-agenten för att bekräfta att datakanalen har öppnats.

Paketformat

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Message Type |
+-+-+-+-+-+-+-+-+
```

Stream Control Transmission Protocol

SCTP är den kraftfulla biten med WebRTC-datakanaler. Det ger alla dessa funktioner i datakanalen:

- Multiplexing av data.
- Pålitlig leverans med en TCP-liknande återsändningsmekanism.
- Alternativ för tillåta en viss paketförlust.
- Inställningar för att undvika trängsel i nätet.
- Flödeskontroll.

För att förstå SCTP kommer vi att utforska protokollet i tre delar. Målet är att du ska veta tillräckligt för att kunna felsöka och själv lära dig detaljer om SCTP efter detta kapitel.

Begrepp

SCTP är ett protokoll med många funktioner. Detta avsnitt kommer endast att gå igenom de delar av SCTP som används av WebRTC. Funktioner i SCTP som inte används av WebRTC inkluderar multi-homing och sökväg (path selection).

Med över tjugo års utveckling bakom sig kan det vara svårt att få en överblick av SCTP.

Association

Association är den term som används för en SCTP-session. Det är tillståndet som delas mellan två SCTP-agenter medan de kommunicerar.

Strömmar

En ström är en dubbelriktad sekvens av användardata. När du skapar en datakanal skapar du bara en SCTP-ström. Varje SCTP Association innehåller en lista med strömmar. Varje ström kan konfigureras med olika tillförlitlighetstyper.

WebRTC låter dig bara konfigurera när strömmen skapas, men SCTP tillåter egentligen att ändra konfigurationen när som helst.

Datagrambaserat

SCTP skickar data som datagram och inte som en byte-ström. Att skicka och ta emot data känns mer som att använda UDP istället för TCP. Du behöver inte lägga till någon extra kod för att skicka flera filer över en ström.

SCTP-meddelanden har inga storleksgränser, till skillnad från UDP. Ett enda SCTP-meddelande kan innehålla flera gigabyte.

Avsnitt

SCTP-protokollet består av avsnitt (chunks). Det finns många olika typer av avsnitt och de används för all kommunikation. Användardata, initialisering av anslutningar, överbelastningskontroll och mer görs via avsnitt.

Varje SCTP-paket innehåller en lista med avsnitt. Så i ett UDP-paket kan du ha flera avsnitt med meddelanden från olika strömmar.

Sändningssekvensnummer

Transmission Sequence Number (TSN) är en global unik identifierare för DATA-avsnitt. Ett DATA-avsnitt är det som innehåller alla meddelanden en användare vill skicka. TSN är viktigt eftersom det hjälper en mottagaren att avgöra om paket har tappats eller om de inte fungerar.

Om mottagaren märker att en TSN saknas, ger den inte datan till användaren förrän den har tagit emot den.

Strömidentifierare

Varje ström har en unik identifierare. När du skapar en datakanal med ett uttryckligt ID skickas det faktiskt precis in i SCTP som strömidentifierare. Om du inte skickar ett ID väljs strömidentifieraren åt dig.

Payload Protocol Identifier

Varje DATA-avsnitt har också en Payload Protocol Identifier (PPID). Detta används för att unikt identifiera vilken typ av data som utbyts. SCTP har många PPID, men WebRTC använder bara följande fem:

- `WebRTC DCEP` (`50`) - DCEP-meddelanden.

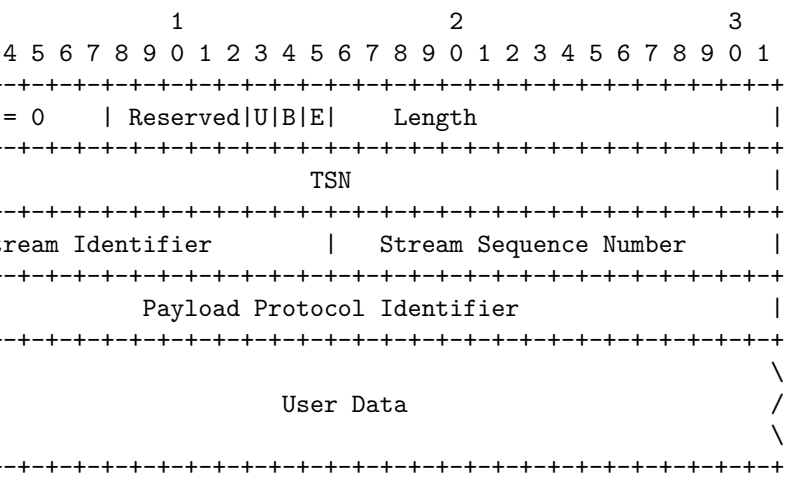
- `WebRTC String` (51) - Strängmeddelanden.
- `WebRTC Binary` (53) - Binära meddelanden.
- `WebRTC String Empty` (56) - Strängmeddelanden med 0 längd.
- `WebRTC Binary Empty` (57) - Binära meddelanden med 0 längd.

Protokoll

Följande är några av de bitar som används av SCTP-protokollet. Detta är inte någon utförlig beskrivning, utan bara de grundläggande strukturerna för att förstå tillståndsmaskinen.

Varje avsnitt börjar med ett `type`-fält. Innan en lista med avsnitt kommer du att finnas en rubrik (header).

DATA-avsnitt



När du skickar data över datakanalen packeteras allt som DATA-avsnitt.

`U`-flaggan är satt om detta är ett oordnat paket. Då kan vi ignorera strömsekvensnumret.

`B` och `E` är början och slutet. Om du vill skicka ett meddelande som är för stort för ett DATA-avsnitt, måste det fragmenteras till flera mindre DATA-avsnitt som skickas var för sig. Flaggorna `B` och `E` och sekvensnummret är hur SCTP kommunicerar detta.

- `B=1`, `E=0` - Första delen av ett fragmenterat användarmeddelande.
- `B=0`, `E=0` - Mellanstycke i ett fragmenterat användarmeddelande.
- `B=0`, `E=1` - Sista delen av ett fragmenterat användarmeddelande.
- `B=1`, `E=1` - Ofragmenterat meddelande.

`TSN` är sändningssekvensnumret. Det är det globala unika identifierare för detta meddelande. Efter 4 294 967 295 DATA-avsnitt börjar räknaren om. Om ett

meddelande fragmenterats till flera DATA-avsnitt, uppdateras sändningssekvensnumret för varje DATA-avsnitt så att mottagaren kan återskapa det ursprungliga meddelandet korrekt.

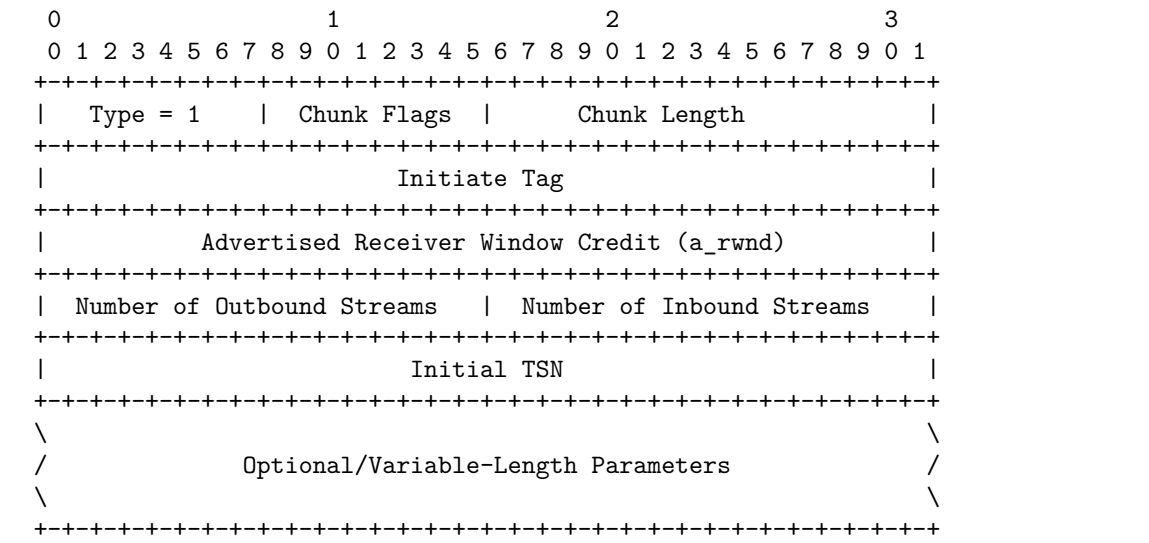
`Stream Identifier` är det unika IDt för strömmen som den här datan tillhör.

`Stream Sequence Number` (strömssekvensnumret) är ett 16-bitars nummer som ökas på för användarmeddelande och ingår i rubriken för varje DATA-avsnitt. Efter 65 535 meddelanden börjar räknaren om på 0. Detta nummer används för att bestämma meddelandets leveransordning till mottagaren om flaggan `U` inte är satt. Det liknar TSN, förutom att strömssekvensnumret bara ökas för varje meddelande som skickas och inte varje enskilt DATA-avsnitt.

`Payload Protocol Identifier` visar vilken typ av data som skickas via den här strömmen. För WebRTC kommer det att vara DCEP, String eller Binary.

`User Data` är den data du skickar. All data du skickar via en WebRTC-datakanal överförs i DATA-avsnitt.

INIT-avsnitt



INIT-avsnittet startar processen att skapa en koppling.

`Initiate Tag` används för att generera kakor. Cookies används för Man-In-The-Middle och Denial of Service-skydd. De beskrivs mer detaljerat i staten maskinsektion.

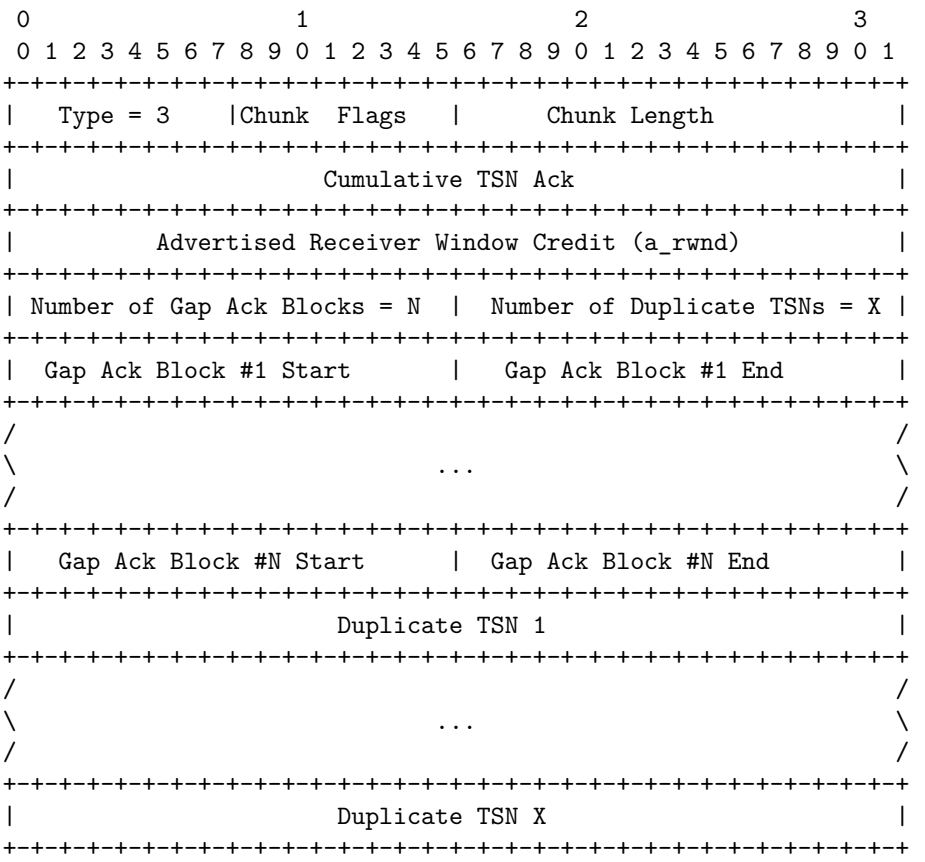
`Advertised Receiver Window Credit` används för SCTPs trängselskontroll. Fältet visar hur stor buffert mottagaren har tilldelat för den här kopplingen.

`Number of Outbound/Inbound Streams` meddelar mottagaren hur många strömmar avsändaren klarar av.

`Initial TSN` är en slumpmässigt vald `uint32` att starta den lokala räknaren på.

`Optional Parameters` gör det möjligt att introducera nya funktioner till SCTP protokollet.

SACK-avsnitt



SACK (Selective Acknowledgment) avsnitt är hur en mottagare meddelar en avsändare att den har fått ett paket. En avsändare kommer att skicka den aktuella DATA-biten om och om igen tills den får en SACK. En SACK gör mer än bara uppdatera TSN dock.

`Cumulative TSN ACK` är det högsta TSN som har mottagits.

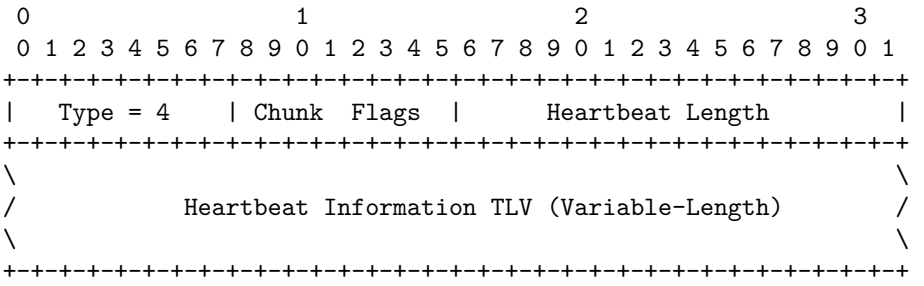
`Advertised Receiver Window Credit` är mottagarens buffertstorlek för det annonserade mottagarfönstret‘. Mottagaren kan ändra detta under sessionen om mer minne blir tillgängligt.

`Ack Blocks` är TSN:er som har tagits emot efter `Cumulative TSN ACK`. Detta används om det finns ett lucka av levererade paket. Låt oss säga DATA-avsnitt

med TSN 100,102, 103 och104 levereras. `Cumulative TSN ACK` skulle vara 100, men `Ack Blocks` kan användas för att meddela avsändaren att den inte behöver skicka 102, 103 eller 104 igen.

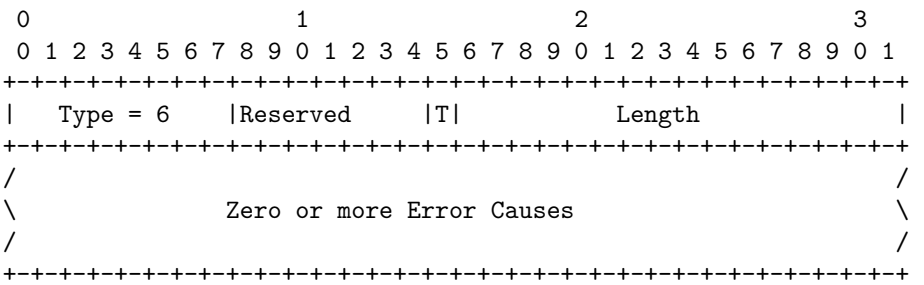
`Duplicate TSN` meddelar avsändaren att den har fått följande DATA-avsnitt mer än en gång.

HEARTBEAT-avsnitt



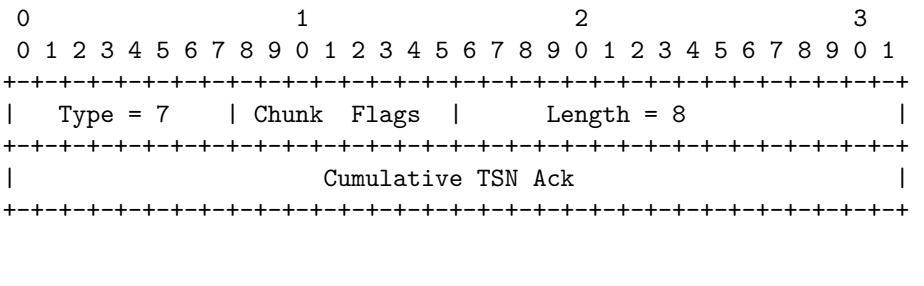
HEARTBEAT-avsnitt används för att kontrollera att mottagaren fortfarande svarar. Användbart om du inte skickar några DATA-avsnitt, men vill behålla en NAT mappning öppen.

ABORT-avsnitt



Ett ABORT-avsnitt stänger plötsligt av kopplingen. Används när ena sidan får ett kritiskt fel. För att avsluta på ett snyggare sätt används SHUTDOWN-avsnitt istället.

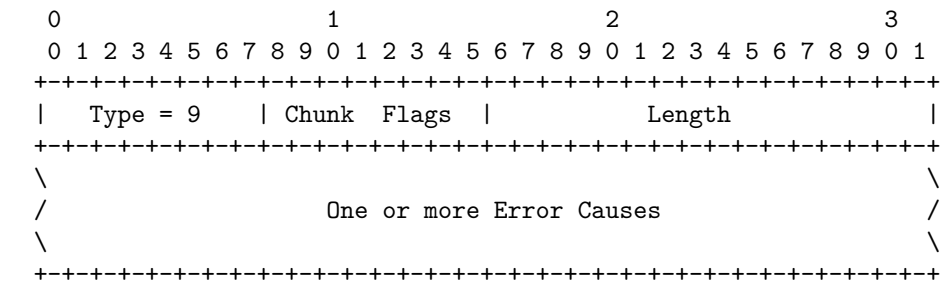
SHUTDOWN-avsnitt



SHUTDOWN-avsnitt startar en kontrollerad avstängning av SCTP-kopplingen. Varje klient informerar mottagaren om den senaste TSN som den skickade. Detta säkerställer att inga paket har tappats. WebRTC gör inte en graciös avstängning av själva SCTP-kopplingen. Du måste stänga ner varje datakanal själv för att hantera det graciöst.

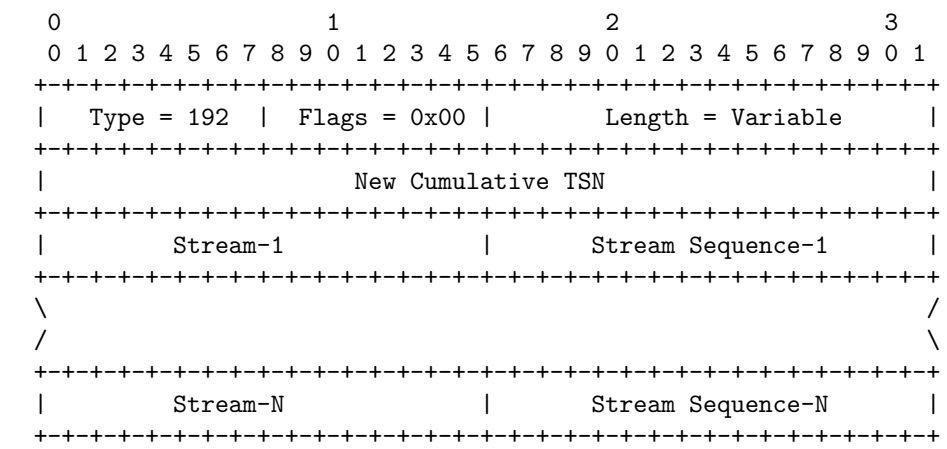
`Cumulative TSN ACK` är den sista TSN som skickades. Varje sida vet att den inte ska stänga ner förrän de har fått DATA-avsnittet med denna TSN.

ERROR-avsnitt



Ett ERROR-avsnitt används för att meddela den andra SCTP klienten att ett icke-kritiskt fel har inträffat.

FORWARD TSN-avsnitt



`FORWARD TSN`-avsnitt flyttar den globala TSN framåt. SCTP gör detta så att du kan hoppa över några paket som du inte bryr dig om längre. Låt oss säga du skickar `10 11 12 13 14 15` och dessa paket endast är giltiga om alla kommer fram. Den här datan måste komma fram i realtid, så om ett paket anländer för sent är det inte användbart.

Om du tappar `12` och `13` finns det ingen anledning att skicka `14` och `15`! SCTP

använder FORWARD TSN-avsnitt för att göra det. Meddelandet låter mottagaren veta att paket 14 och 15 inte kommer att skickas längre.

New Cumulative TSN är den nya TSN:en för anslutningen. Alla paket innan detta TSN kommer att slängas.

Stream och Stream Sequence används för att hoppa Stream Sequence Number antal avsnitt framåt. Gå tillbaka till DATA-avsnitt för betydelsen av detta fält.

Tillståndsmaskinen för SCTP

Det finns några intressanta delar av SCTPs tillståndsmaskin. WebRTC använder inte alla funktioner i SCTP-tillståndsmaskinen, så vi har uteslutit vissa delar. Vi har också förenklat vissa komponenter för att göra dem begripliga på egen hand.

Anslutningsetableringsflöde

INIT och INIT ACK avsnitt används för att utbyta funktioner och konfigurationer mellan varje klient. SCTP använder en kaka (cookie) under handskakningen för att validera den kollega den kommunicerar med. Detta för att säkerställa att handskakningen inte avlyssnas och för att förhindra DoS-attacker.

INIT ACK avsnittet innehåller kakan. Kakan returneras sedan till dess skapare med COOKIE ECHO. Om cookieverifiering lyckas skickas ett COOKIE ACK och DATA-avsnitt är redo att börja skickas.

Anslutning av kopplingen

SCTP använder SHUTDOWN avsnittet. När en agent får ett SHUTDOWN-avsnitt kommer den att vänta tills den fått det begärda Cumulative TSN ACK. Detta gör det möjligt för en användare att se till att all data levereras även om anslutningen har hög paketförlust.

Keep-Alive mekanism

SCTP använder HEARTBEAT REQUEST och HEARTBEAT ACK avsnitt för att hålla anslutningen vid liv. Dessa meddelanden skickas på ett konfigurerbart intervall. SCTP använder sig av exponentiell backoff om paketet inte har kommit fram.

HEARTBEAT-avsnitt innehåller också ett tidsvärde. Detta gör det möjligt för två kopplingar att beräkna tur-och-retur-tiden mellan två klienter.

Hur WebRTC används

Nu när du vet hur WebRTC fungerar är det dags att bygga något med det! Detta kapitel utforskar vad människor har byggt med WebRTC och hur de bygger det. Du kommer att lära dig alla intressanta saker som är händer med WebRTC.

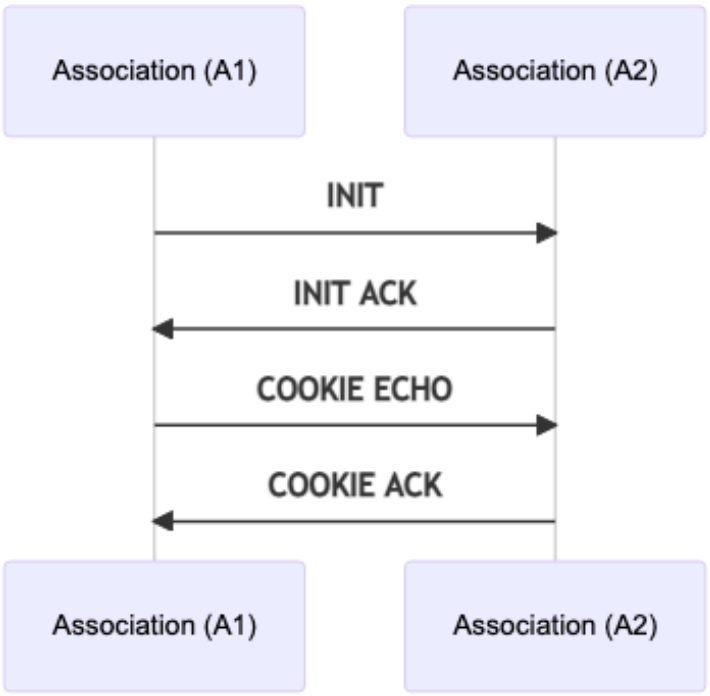


Figure 13: Uppkoppling

Kraften i WebRTC kostar. Att bygga WebRTC-tjänster med produktionskvalitet är utmanande. Detta kapitel kommer att försöka förklara dessa utmaningar innan du drabbas av dem.

Användningsfall

Många tror att WebRTC bara är en teknik för konferenser i webbläsaren, men det finns så mycket mer man kan använda det till! WebRTC används på många olika sätt och nya användningsfall dyker upp hela tiden. I detta kapitel kommer vi att lista några vanliga användningsfall och hur WebRTC revolutionerar dem.

Konferenser

Konferenser är det ursprungliga användningsfallet för WebRTC. Protokollet innehåller några nödvändiga funktioner som inget annat protokoll erbjuder i webbläsaren. Du kan bygga ett konferenssystem med WebSockets och det kan fungera under optimala förhållanden. Men om du vill ha något som tål alla störningar som finns i våra nätverk är WebRTC det bästa valet.

WebRTC ger överbelastningskontroll och adaptiv bithastighet för media. När villkoren för nätverket ändras kommer användarna fortfarande att få bästa möjliga upplevelse. Utvecklare behöver inte skriva någon ytterligare kod för att hantera dessa villkor.

Deltagare kan skicka och ta emot flera strömmar. De kan också lägga till och ta bort dessa strömmar när som helst under samtalet. Codecs förhandlas fram också. All denna funktion tillhandahålls redan av webbläsaren, ingen ny kod behöver skrivas av utvecklaren.

Konferenser drar också nytta av datakanaler. Användare kan skicka metadata eller dela dokument. Du kan skapa flera strömmar och konfigurera dem om du prioriterar prestanda mer än tillförlitlighet.

Utsändning

Många nya projekt relaterade till utsändning använder sig av WebRTC. Protokollet har mycket att erbjuda för både utgivare och konsumenter av media.

WebRTC i webbläsaren gör det enkelt för användare att publicera video. Det tar bort kravet för användare att ladda ner en ny klient. Alla plattformar som har en webbläsare kan publicera video. Utgivare kan sedan skicka flera spår och ändra eller ta bort dem när som helst. Detta är en enorm förbättring jämfört med äldre protokoll som bara tillät ett ljud eller videospår per anslutning.

WebRTC ger utvecklare större kontroll över avvägningen mellan latens och kvalitet. Det kan vara viktigare att latensen aldrig överstiger ett visst värde, och du kanske är villig att istället visa lite sämre videokvalité. Du kan konfigurera dekodern att spela upp media så snart den tagits emot. Med andra protokoll

som körs över TCP är det inte lika enkelt. I webbläsaren kan du begära data och det är allt.

Fjärranslutning

Fjärråtkomst är när du ansluter till en annan dator via WebRTC. Du kan ha fullständig kontroll över hela datorn, eller kanske bara en applikation på den. En vanlig tillämpning är att köra tunga beräkningsuppgifter som den lokala hårdvaran inte klarar av. Som att köra ett nytt videospel, eller CAD-programvara. WebRTC har revolutionerat det här området på tre sätt.

WebRTC kan användas för att ansluta till en server som inte har någon publik adress. Med NAT Traversal kan du komma åt en dator som endast är tillgänglig via STUN. Detta är bra för säkerhet och anonymitet. Dina användare behöver inte skicka video via någon publik server. NAT Traversal gör också distributionen enklare. Du behöver inte oroa dig om att öppna portar eller ställa in et statiskt IP i förväg.

Datakanaler är också mycket kraftfulla i det här scenariot. De kan konfigureras så att endast de senaste uppgifterna accepteras. Med TCP riskerar du att stöta på head-of-line blockering. Ett gammalt musklick eller tangenttryckning kan komma för sent och blockera nyare händelser. WebRTCs datakanaler är utformade för att hantera detta och kan konfigureras för att inte försöka skicka förlorade paket igen. Du kan också mäta mottrycket (backpressure) och se till att du inte skickar mer data än ditt nätverk klarar av.

Att WebRTC är tillgängligt i webbläsaren har gjort allt betydligt enklare. Du behöver inte ladda ner någon egen klient för att starta ett samtal. Fler och fler kunder kommer med WebRTC-paket, till och med smarta TV-apparater har fullt stöd för webbläsare nu.

Fildelning och undvika censur

Fildelning och att undvika censur är väldigt olika problem. WebRTC kan dock lösa båda problemen. Den gör båda lätt tillgängliga och svårare att blockera.

Det första problemet som WebRTC löser är att du redan har en klient. Om du vill gå med i ett nätverk för fildelning måste du ladda ner klienten. Även om nätverket distribueras måste du fortfarande få klienten först. I ett begränsat nätverk kommer nedladdningar ofta att blockeras. Och även om du kan ladda ner den kanske du inte kan installera och köra klienten. WebRTC är redan tillgängligt i alla moderna webbläsare.

Det andra problemet som WebRTC löser är att din trafik blockeras. Om du använder ett protokoll som bara är gjort för fildelning eller f;r att kringgå censur, är det mycket lättare att blockera det. Eftersom WebRTC är ett mer generellt protokoll kan blockering av det påverka vem som helt. Blockering av WebRTC kan till exempel hindra andra användare i nätverket från att koppla upp sig till ett konferenssamtal.

Internet of Things

Internet of Things (IoT) täcker några olika användningsfall. För många betyder detta nätverksanslutna säkerhetskameror. Med WebRTC kan du strömma videon till en annan WebRTC klient som till exempel din telefon eller en webbläsare. Ett annat användningsfall är att enheter ansluter och utbyter sensordata. Du kan ha två enheter i ditt lokala nätverk utbyta temperatur, ljud eller ljus-mätvärden.

WebRTC har en enorm integritetsfördel här jämfört med äldre videoströmprotokoll. Eftersom WebRTC stöder P2P-anslutning kan kameran skicka videon direkt till din webbläsare. Det finns ingen anledning att din video ska skickas via en tredjepartsserver. Även när video är krypterad kan en angripare göra antaganden baserat på samtalens metadata.

Interoperabilitet är en annan fördel när det kommer till IoT. WebRTC finns implementerat för många olika språk; C#, C++, C, Go, Java, Python, Rust och TypeScript. Det betyder att du kan använda det språk som fungerar bäst för dig. Du behöver inte heller använda företagsspecifika protokoll eller proprietära format för att koppla ihop dina klienter.

Översättning av medieprotokoll

Tänk dig att du har befintlig hårdvara och programvara som producerar video, men du kan inte uppgradera den just nu. Att förvänta sig att användare ska ladda ner en proprietär klient för att kunna se videon är onödigt frustrerande. En enklare lösning är att sätta upp en WebRTC-bro. Bron översätter mellan det gamla och nya protokollen så att användarna kan använda webbläsare för att se videon från din äldre installation.

Många av de format som utvecklare sätter upp broar mot med använder samma protokoll som WebRTC. SIP exponeras vanligtvis via WebRTC och tillåter användare att ringa telefonsamtal direkt från webbläsaren. RTSP används i många äldre säkerhetskameror. De använder båda samma underliggande protokoll (RTP och SDP) eftersom de kräver väldigt lite beräkningskraft att köra. WebRTC-bron krävs bara för att lägga till eller ta bort saker som är WebRTC-specifika.

Översättning av dataprotokoll

En webbläsare kan bara tala en begränsad uppsättning protokoll. Du kan använda HTTP, WebSockets, WebRTC och QUIC. Om du vill ansluta till något annat behöver du översätta protokollet. En protokollbro är en server som omvandlar data trafik till något som webbläsaren kan förstå. Ett populärt exempel är att använda SSH från din webbläsare för att komma åt en server. WebRTCs datakanaler har två fördelar framför konkurrenterna.

WebRTCs datakanaler möjliggör opålitlig och oordnad leverans. I fall där låg latens är avgörande är detta ett måste. Du vill inte att ny data blockeras av gammal data, så kallad head-of-line blocking. Tänk dig att du spelar en snabbt

multiplayer förstapersonsspel. Bryr du dig verkligen om var en annan spelare var för två sekunder sedan? Om datan inte kom fram i tid är det meningslöst att fortsätta försöka skicka den. Otillförlitlig och oordnad leverans låter dig få datan så snart den anländer.

Datakanaler ger också information om belastningen i nätet. Det berättar om du skickar data snabbare än vad din anslutning kan hantera. Du har två lösningar att välja på när det händer. Datakanalen kan antingen konfigureras för att buffra och leverera data lite senare, eller så kan du strunta i data som inte har kommit fram i tid.

Teleoperation

Teleoperation är när man fjärrstyr en enhet via WebRTC-datakanaler och skickar tillbaka videon via RTP. Utvecklare kör bilar på avstånd via WebRTC idag! Det används för att kontrollera robotar på byggarbetsplatser och för att leverera paket. Att använda WebRTC för dessa problem är praktiskt av två skäl.

Att WebRTCs är så lättillgängligt gör det enkelt att ge användare kontrollen. Allt användaren behöver är en webbläsare och något att styra med. Webbläsare har till och med stöd för joysticks och gamepads. WebRTC tar helt enkelt bort behovet av att installera ytterligare en klient på användarens enhet.

Distribuerad CDN

Distribuerade CDN:er är en typ av fildelning. Filerna som distribueras konfigureras istället av CDN-operatören. När användare ansluter sig till CDN-nätverket de kan ladda ner och dela de tillåtna filerna. Användare får samma fördelar som fildelning.

Dessa CDN fungerar bra när du är på ett kontor som har en dålig internetuppkoppling, men ett snabbt lokalt nätverk. Du kan låta en användare ladda ner en video och dela det sedan med alla andra. Eftersom alla inte försöker hämta samma fil via det externa nätverket kommer överföringen att gå mycket snabbare.

WebRTC Topologier

WebRTC är ett protokoll för att ansluta två klienter, så hur ansluter utvecklare hundratals människor på en gång? Det finns några olika sätt du kan göra det, och de har alla olika för och nackdelar. Dessa lösningar faller i stort sett i två kategorier; peer-to-peer eller klient/server. WebRTCs flexibilitet tillåter oss att implementera båda.

En till en

En-till-en är den första anslutningstypen du använder med WebRTC. Du ansluter två WebRTC-agenter direkt och de kan skicka dubbelriktad media och data.

Anslutningen ser ut så här.

Figure 14: En-till-en

Full Mesh

Full mesh är svaret om du vill bygga ett konferenssamtal eller ett multiplayer-spel. I denna topologi upprättar varje användare en anslutning med alla andra användare direkt. Detta gör att du kan bygga din applikation, men det kommer med några nackdelar.

I en Full Mesh kopplas varje användare direkt mot alla andra. Det betyder att du måste koda och ladda upp video oberoende för varje medlem i samtalet. Nätverksförhållandena mellan varje anslutning kommer att vara olika, så du kan inte återanvända samma video. Felhanteringen är också svår i den här typen av lösningar. Du måste noga överväga om du har helt tappat din anslutning eller bara anslutning med en av de andra klienterna.

På grund av dessa problem används ett Full Mesh bäst för små grupper. För större grupper är en klient/server topologi bäst.

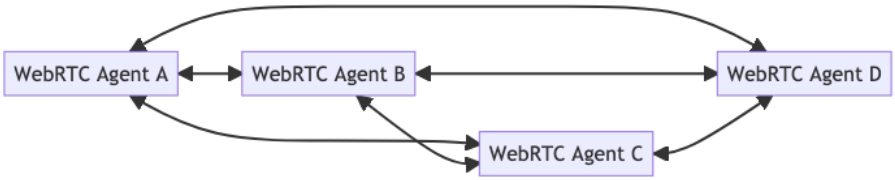


Figure 15: Full mesh

Hybrid Mesh

Hybrid Mesh är ett alternativ till Full Mesh som kan lindra några av problemen mett ett Full Mesh. I en Hybrid Mesh-anslutning upprättas inte mellan varje användare. Istället vidarebefordras media genom andra klienter i nätverket. Detta betyder att avsändaren av media inte behöver använda lika mycket bandbredd för att distribuera media.

Detta har dock några nackdelar. I denna konfiguration har den ursprungliga skaparen av media ingen aning vem dess video skickas till, eller om den ens kom fram. Du kommer också att öka latensen för varje hopp i ditt Hybrid Mesh-nätverk.

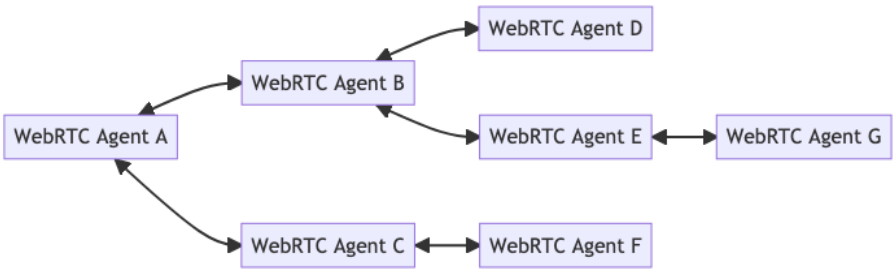


Figure 16: Hybrid mesh

Selektiv vidarebefordringsenhet

En SFU (Selective Forwarding Unit) löser också problemen med ett Full Mesh, men på ett helt annat sätt. En SFU implementerar en klient/server topologi istället för P2P. Varje WebRTC-peer ansluter till SFU:en och laddar upp sin media. SFU:en vidarebefordrar sedan denna media till varje ansluten klient.

Med en SFU behöver varje WebRTC-agent bara koda och ladda upp sin video en gång. Arbetet att distribuera den till alla tittare ligger på SFUn. Anslutning till en SFU är också mycket enklare än P2P. Du kan köra en SFU på en publik IP adress, vilket gör det mycket lättare för kunder att ansluta. Du behöver inte oroa dig för NAT Mappings. Du måste fortfarande se till att din SFU är tillgänglig via TCP (antingen via ICE-TCP eller TURN).

Att bygga en enkel SFU kan göras på en helg. Att bygga en bra SFU som kan hantera alla typer av klienter är näst intill omöjligt. Att justera trängselkontroll, felkorrigering och prestanda är ett arbete som aldrig tar slut.

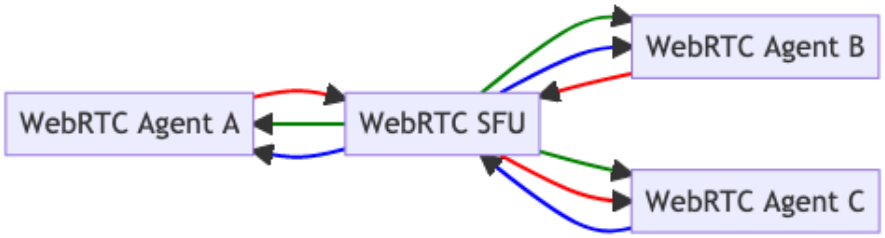


Figure 17: Selektiv vidarebefordringsenhet

MCU

En MCU (Multi-point Conferencing Unit) är en klient/server-topologi som en SFU, men den sätter ihop videoströmmarna. Istället för att distribuera utgående media omodifierad kodar den om alla inkommande strömmar till en utgående videoström.

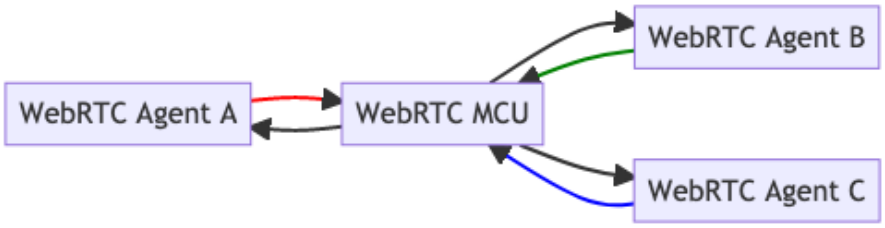


Figure 18: Multi-point Conferencing Unit

#Felsökning Felsökning av WebRTC kan vara en trixig uppgift. Det finns många rörliga delar, och de kan alla gå sönder oberoende av varandra. Om du inte är försiktig kan du förlora veckor på att kolla på fel saker. När du äntligen hittar den del som är trasig måste du lära dig mer för att förstå varför.

Detta kapitel kommer att få dig att tänka på rätt sätt för att felsöka WebRTC. Det kommer att visa dig hur du bryter ner problemet. När vi känner till problemet ger vi en snabb översikt av de mest populära felsökningsverktygen.

Isolera problemet

När du felsöker måste du isolera var problemet kommer ifrån. Börja från början av...

Signalfel TODO!

Nätverksfel Testa din STUN-server med netcat:

1. Förbered ett **20-byte** bindningsförfråganspaket:

   ```
   echo -ne "\x00\x01\x00\x00\x21\x12\xA4\x42TESTTESTTEST" | hexdump -C
   00000000  00 01 00 00 21 12 a4 42  54 45 53 54 54 45 53 54  |....!..BTESTTEST|
   00000010  54 45 53 54                                       |TEST|
   00000014
   ```

   Förklaring:

   - `00 01` är meddelandetypen.
   - `00 00` är längden på datan.
   - `21 12 a4 42` är den magiska kakan.
   - och `54 45 53 54 54 45 53 54 54 45 53 54` (I ASCII: `TESTTESTTEST`) är ett 12-byte långt transaktions-ID.

2. Skicka meddelandet och vänta på det **32 byte** stora svaret:

   ```
   stunserver=stun1.l.google.com;stunport=19302;listenport=20000;echo -ne "\x00\x01\x00\x0
   00000000  01 01 00 0c 21 12 a4 42  54 45 53 54 54 45 53 54  |....!..BTESTTEST|
   00000010  54 45 53 54 00 20 00 08  00 01 6f 32 7f 36 de 89  |TEST. ....o2.6..|
   ```

```
00000020
```

Förklaring:

- `01 01` är meddelandetypen.
- `00 0c` är längden på datan, i det hör fallet 12 i decimaltal.
- `21 12 a4 42` är den magiska kakan.
- och `54 45 53 54 54 45 53 54 54 45 53 54` (I ASCII: `TESTTESTTEST`) är ett 12-bitars transaktions-ID
- `00 20 00 08 00 01 6f 32 7f 36 de 89` de 12-bitarna med data, som översatt blir:
  - `00 20` är typen: `XOR-MAPPED-ADDRESS`.
  - `00 08` är längden på värdet, i det här fallet 8 bytes
  - `00 01 6f 32 7f 36 de 89` är själva värdet, som översatt blir:
    * `00 01` adresstyp (IPv4)
    * `6f 32` den XOR-mappade porten
    * `7f 36 de 89` den XOR-mappede IP-adressen

Avkodning av det XOR-mappade avsnittet är lite krångligt, men vi kan lura stun-servern att utföra en dummy-XOR-mappning genom att skicka en (ogiltig) dummy-magisk cookie inställd på `00 00 00 00`:

```
stunserver=stun1.l.google.com;stunport=19302;listenport=20000;echo -ne "\x00\x01\x00\x00\x0
00000000  01 01 00 0c 00 00 00 00  54 45 53 54 54 45 53 54  |........TESTTEST|
00000010  54 45 53 54 00 01 00 08  00 01 4e 20 5e 24 7a cb  |TEST......N ^$z.|
00000020
```

XOR på en magiska kakan med bara nollor ändrar inget, så porten och adressen kommer att vara i klartext i svaret. Detta fungerar inte alltid, eftersom vissa routrar manipulerar de passerande paketen, fusk på IP-adressen. Om vi tittar på det returnerade värdet (senaste åtta byten):

- `00 01 4e 20 5e 24 7a cb` data värdet, som översatt blir:
  - `00 01` adresstyp (IPv4)
  - `4e 20` portwn, i det hör fallet 20000 i decimal tal.
  - `5e 24 7a cb` är IP adressen, `94.36.122.203` i mer lättläst form.

Säkerhetsfel

Mediefel

Datafel

Tools of the trade

netcat (nc) netcat är ett nätverksverktyg för kommandoraden som kan läsa och skriva till nätverksanslutningar via TCP eller UDP. Det är vanligtvis tillgängligt som kommandot `nc`.

tcpdump tcpdump är ett datanätverkspaketanalysverktyg för kommandoraden.

Vanliga kommandon: - Fånga UDP-paket till och från port 19302, skriv ut en hexdump av paketinnehållet:

```
sudo tcpdump 'udp port 19302' -xx
```

- Samma, men spara istället paketen i en PCAP-fil (packet capture) för senare inspektion:

  ```
  sudo tcpdump 'udp port 19302' -w stun.pcap
  ```

  PCAP-filen kan öppnas med Wireshark-applikationen: `wireshark stun.pcap`

wireshark

webrtc-internals Chrome har en inbyggd WebRTC-statistiksida som du hittar på chrome://webrtc-internals.

Latens

Hur vet du om du har hög latens? Du kanske har märkt att din video släpar efter, men vet du exakt hur mycket efter den är? För att kunna minska latensen måste du börja med att mäta den först.

Sann latens ska mätas hela vägen. Det betyder inte bara latensen för nätverket mellan avsändaren och mottagaren, utan den kombinerade latensen för kamera, kodande av video, överföring, mottagning, avkodning och visning, samt eventuell a köer mellan dessa steg.

Total latens kan inte beräknas som en summa av latensen för varje komponent.

Även om du teoretiskt kan mäta latensen för alla komponenter i en live videoöverföring separat och sedan lägga hop dem, i praktiken kommer åtminstone vissa komponenter antingen att vara oåtkomliga för instrumentering eller ge ett helt annat resultat när de mäts utanför överföringen. Olika buffrings-effekter mellan steg i video pipelinen, nätverkstopologi och automatiska kamera justeringar är bara några exempel på komponenter som påverkar den faktiska latensen.

Den inneboende latensen för varje komponent i ditt livesändningssystem kan förändras och påverka nedströms komponenter. Även innehållet i den inspelade videon påverkar latensen. Till exempel krävs många fler bitar för komplexa bilder såsom trädgrenar, jämfört med en klarblå himmel som innehåller mycket mindre information. En kamera med automatisk exponering aktiverad kan ta *mycket* längre tid än de förväntade 33 millisekunderna för att ta en bild, även om inspelningshastigheten är inställd på 30 bilder per sekund. Överföring över nätverket, särskilt mobila nät, ändras också ofta på grund av förändrad efterfrågan och kapacitet. Fler användare introducerar mer störningar i luften. Din fysiska

plats (områden med dålig täckning) och flera andra faktorer ökar paketförlust och latens. Vad händer när du skickar ett paket till ett nätverksinterface, till exempel en WiFi-adapter eller ett LTE-modem? Om paketet inte kan skickas omedelbart köas det upp, ju längre kö desto mer latens introducerar ett sådant nätverksinterface.

Manuell mätning från start till slut

När vi pratar om start-till-slut-latens menar vi tiden mellan en händelse som inträffar och när den observeras, det vill säga att bilden visas på skärmen.

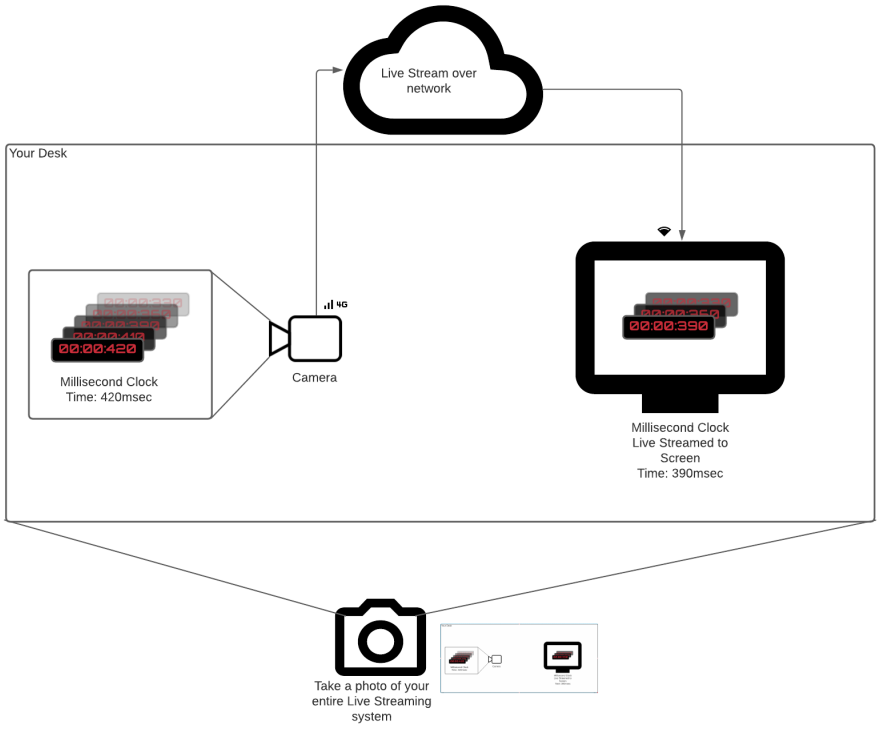
```
EndToEndLatency = T(observe) - T(happen)
```

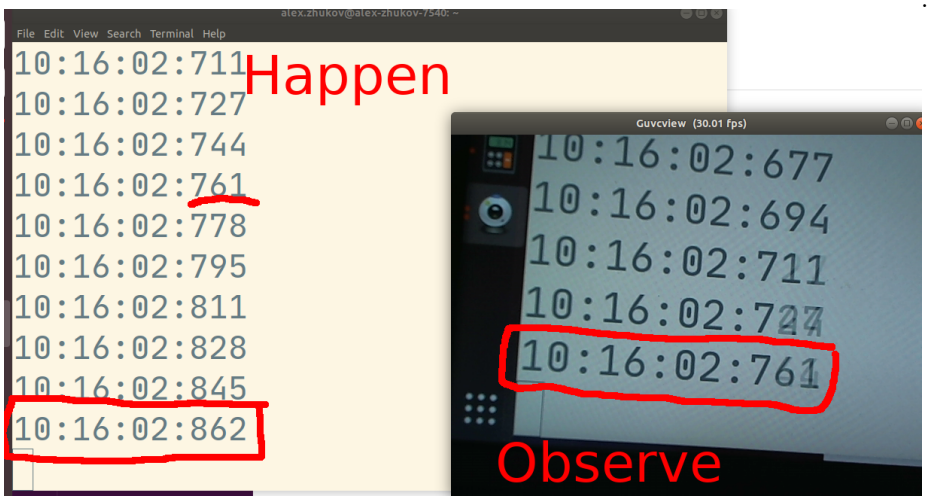
Ett naivt tillvägagångssätt är att mäta tiden när en viss händelse sker och subtrahera den från tiden vid när man kan se videon av händelsen. Men när precisionen är nere på millisekunder blir tidssynkronisering ett problem. Att försöka synkronisera klockor över distribuerade system är oftast meningslöst, även ett litet fel i tidssynkronisering ger opålitliga mätningar.

En enkel lösning på klocksynkroniseringsproblemet är att använda samma klocka, att placera sändare och mottagare i samma referensram.

Tänk dig att du verkligen har en tickande millisekund-klocka eller någon annan händelsekälla. Du vill mäta latens i ett system som live streamar klockan till en skärm genom att peka en kamera mot den. Ett uppenbart sätt att mäta tiden mellan att millisekundtimern tickar (`Thappen`) och klockans bild visas på skärmen (`Tobserve`) är följande: - Rikta kameran mot klockan. - Skicka videon till en mottagare som är på samma fysiska plats. - Ta en bild (använd din telefon) av klockan och den mottagna videon på skärmen. - Subtrahera två gånger.

Det är den mest riktiga mätningen för start-till-slut-latens. Den innehåller alla komponenters latens (kamera, kodek, nätverk, avkodare) och är inte beroende av någon klocksynkronisering.





På bilden ovan är uppmätta start-till-slut-latensen 101 ms. Tiden som syns just nu är 10:16:02.862, men observatören av live-streaming-systemet ser 10:16:02.761.

Automatisk start-till-slut-mätning

I skrivande stund (maj 2021) diskuteras WebRTC-standarden för start-till-slut-latens aktivt. Firefox implementerade en uppsättning API:er för att låta användare skapa automatisk latensmätning över standard WebRTC API:et. Men i det här kapitlet diskuterar vi det mest kompatibla sättet att automatiskt mäta latens.

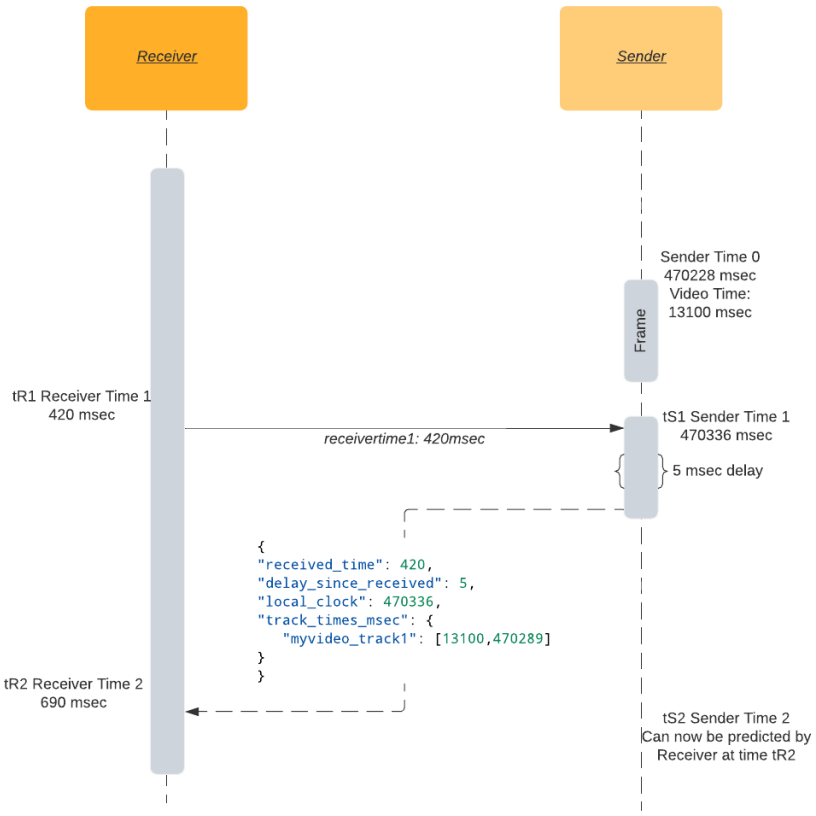


Figure 19: NTP Style Latency Measurement

Tur-och-returtid i ett nötskal: Jag skickar dig min tid `tR1`, när jag får tillbaka min `tR1` vid tiden `tR2` vet jag att tur-och-returtiden är `tR2 - tR1`.

Tänk dig att du har en kommunikationskanal mellan sändare och mottagare (t.ex. en DataChannel. Då kan mottagaren modellera avsändarens monotona klocka genom att följa nedanstående steg: 1. Vid tidpunkten `tR1` skickar mottagaren ett meddelande med sin lokala klocktid. 2. När den tas emot av sändaren med lokal tid `tS1`, svarar sändaren med en kopia av `tR1` samt sändarens `tS1` och sändarens

video-tid `tSV1`. 3. Vid tidpunkten `tR2` hos mottagaren beräknas tur-och-retur-tiden genom att subtrahera meddelandets sändnings- och mottagningstider: `RTT = tR2 - tR1`. 4. Tur-och-retur-tid (`RTT`) tillsammans med avsändarens lokala tid `tS1` är tillräckligt för att göra en uppskattning av avsändarens monotona klocka. Den aktuella tiden hos avsändaren vid tiden `tR2` bör vara lika med `tS1` plus hälften av tur-och-retur-tiden. 5. Avsändarens lokala klocka `tS1` tillsammans med videospårets tidsstämpel `tSV1` och tur-och-retur-tiden `RTT` räcker därför för att synkronisera mottagarens video med avsändarens video.

Nu när vi vet hur mycket tid som har gått sedan senaste video-bilden (`tSV1`), kan vi göra en ungefärlig uppskattning av latensen genom att dra bort klockstämpeln hos video-bilden som visas just nu från den uppskattade tiden:

```
expected_video_time = tSV1 + time_since(tSV1)
latency = expected_video_time - actual_video_time
```

Metodens nackdel är att den inte inkluderar själva kamerans latens. De flesta videosystem anser att tidsstämpeln för bildtagning är den tid då kameran skickat bilden till minnet, vilket kommer att vara några ögonblick efter att själva kamera sensorn tog emot bilden.

Exempel på latensuppskattning I exemplet nedan öppnas en datakanal kallad `latency` hos mottagaren, som periodiskt skickar tidsstämplar till sändaren. Sändaren svarar tillbaka med ett JSON-meddelande. Mottagaren beräknar latensen baserat på det meddelandet.

```
{
    "received_time": 64714,       // Timestamp sent by receiver, sender reflects the timesta
    "delay_since_received": 46,   // Time elapsed since last `received_time` received on sen
    "local_clock": 1597366470336, // The sender's current monotonic clock time.
    "track_times_msec": {
        "myvideo_track1": [
            13100,        // Video frame RTP timestamp (in milliseconds).
            1597366470289 // Video frame monotonic clock timestamp.
        ]
    }
}
```

Öppna en data-kanal hos mottagaren:

```
dataChannel = peerConnection.createDataChannel('latency');
```

Skicka mottagarens tid `tR1` med jämna mellanrum. Valet av 2 sekunder har ingen specifik betydelse:

```
setInterval(() => {
    let tR1 = Math.trunc(performance.now());
    dataChannel.send("" + tR1);
}, 2000);
```

Hantera inkommande meddelanden från mottagaren hos sändaren:

```
// Assuming event.data is a string like "1234567".
tR1 = event.data
now = Math.trunc(performance.now());
tSV1 = 42000; // Current frame RTP timestamp converted to millisecond timescale.
tS1 = 1597366470289; // Current frame monotonic clock timestamp.
msg = {
    "received_time": tR1,
    "delay_since_received": 0,
    "local_clock": now,
    "track_times_msec": {
        "myvideo_track1": [tSV1, tS1]
    }
}
dataChannel.send(JSON.stringify(msg));
```

Hantera inkommande meddelande från sändaren och visa den beräknade latensen i konsolen:

```
let tR2 = performance.now();
let fromSender = JSON.parse(event.data);
let tR1 = fromSender['received_time'];
let delay = fromSender['delay_since_received']; // How much time that has passed between the
let senderTimeFromResponse = fromSender['local_clock'];
let rtt = tR2 - delay - tR1;
let networkLatency = rtt / 2;
let senderTime = (senderTimeFromResponse + delay + networkLatency);
VIDEO.requestVideoFrameCallback((now, framemeta) => {
    // Estimate current time of the sender.
    let delaySinceVideoCallbackRequested = now - tR2;
    senderTime += delaySinceVideoCallbackRequested;
    let [tSV1, tS1] = Object.entries(fromSender['track_times_msec'])[0][1]
    let timeSinceLastKnownFrame = senderTime - tS1;
    let expectedVideoTimeMsec = tSV1 + timeSinceLastKnownFrame;
    let actualVideoTimeMsec = Math.trunc(framemeta.rtpTimestamp / 90); // Convert RTP timeb
    let latency = expectedVideoTimeMsec - actualVideoTimeMsec;
    console.log('latency', latency, 'msec');
});
```

Actual video time in browser

> `<video>.requestVideoFrameCallback()` tillåter webbutvecklare att få ett meddelande när en bild har är tillgänglig att visas.

Fram till mycket nyligen (maj 2020), var det nästan omöjligt att på ett tillförlitligt sätt få en tidsstämpel för den visade bilden i en webbläsare. Lösningar baserade på `video.currentTime` fanns, men de var inte särskilt exakta. Utveck-

lare från både Chrome och Mozilla stödde införandet av en ny W3C-standard, `HTMLVideoElement.requestVideoFrameCallback()`, som lägger till ett nytt API för att komma åt den aktuella bildens tidsstämpel. Även om tillägget låter trivialt, har det möjliggjort flera avancerade medieapplikationer via nätet som kräver ljud- och videosynkronisering. För WebRTC inkluderar API:et fältet `rtpTimestamp` i meddelandet, RTP-tidsstämpeln associerad med den aktuella bilden. Detta fält finns bara för WebRTC-applikationer, annars saknas det.

Tips för att felsöka latensproblem

Eftersom felsökning sannolikt kommer att påverka den uppmätta latensen är den allmänna regeln att förenkla dit system till minsta möjliga som fortfarande kan reproducera problemet. Ju fler komponenter du kan ta bort, desto lättare blir det att ta reda på vilken komponent som orsakar latensproblemet.

Kameralatens Beroende på kamerainställningar kan kamerans latens variera. Kontrollera inställningar för automatisk exponering, autofokus och automatisk vitbalans. Alla "auto"-funktionerna på webbkameror tar lite extra tid för att analysera den tagna bilden innan den är tillgänglig för WebRTC-stacken.

Om du kör Linux kan du använda kommandoradsverktyget `v4l2-ctl` för att ändra kamerainställningarna:

```
# Stäng av autofocus:
v4l2-ctl -d /dev/video0 -c focus_auto=0
# Sätt fokus till max:
v4l2-ctl -d /dev/video0 -c focus_absolute=0
```

Du kan också använda det grafiska verktyget `guvcview` för att kontrollera och justera kamerainställningarna.

Kodarens latens De flesta moderna kodare kommer att buffra ett antal bilder innan de kodas. Deras högsta prioritet är att hitta en balans mellan kvaliteten på den producerade bilden och hur mycket data varje bild använder (bitrate). Multipass-kodning är ett extremt exempel där kodaren helt bortser från latensen. Under det första passet tar den in hela videon och först därefter börjar utmatning av bilder.

Men med rätt inställningar är det möjligt att nå latens lägre än den önskade bildfrekvensen. Se till att din kodare inte använder för många I-bilder eller förlitar dig på B-bilder. Varje codecs inställningar för latensinställning är olika, men för x264 rekommenderar vi att du använder `tune=zerolatency` och `profile=baseline` för få så låg latens som möjligt.

Nätverksfördröjning Nätverksfördröjningar är nog det kan göra minst åt, förutom att uppgradera till en bättre internetanslutning. Nätverksfördröjningar är ungefär som vädret - du kan inte stoppa regnet, men du kan kolla prognosen och

ta med ett paraply. WebRTC mäter nätverksförhållanden med millisekundprecision. Viktiga mätvärden är: - Tur-och-returtid - Paketförlust och omskickande av paket.

Tur-och-returtid

WebRTC-stacken har en inbyggd nätverksmätningsmekanism för tur-och-returtid. En tillräckligt bra uppskattning av latens är hälften av RTT. Det förutsätter att det tar samma tid att skicka och ta emot ett paket, vilket inte alltid är fallet. Tur-och-returtid sätter den lägre gränsen för start-till-slut-latens. Dina bilder når inte mottagaren snabbare än `RTT/2`, oavsett hur snabbt din kamera skickar bilder till kodaren.

Den inbyggda RTT-mekanismen är baserad på speciella RTCP-paket som kallas avsändar-/mottagarrapporter. Avsändaren skickar sin tidsavläsning till mottagaren, mottagaren återspeglar i sin tur samma tidsstämpel till avsändaren. Därmed vet avsändaren hur mycket tid det tog för paketet att skickas till mottagaren och tillbaka. Se kapitlet Avsändar-/mottagarrapporter för mer information om RTT-mätning.

Paketförlust och omskickande av paket

Både RTP och RTCP är protokoll baserade på UDP som inte har någon garanti för varken ordning, leverans eller duplicering. Allt ovanstående fall dyker upp när du använder WebRTC-applikationer. En osofistikerad avkodarimplementering förväntar sig att alla paket i en bild ska levereras för att avkodaren ska sätta ihop bilden igen. Vid paketförlust kan avkodningsartefakter uppstå om paket med en P-bild går förlorade. Om en I-bild-paket tappas får alla följande bilder antingen grava artefakter, eller kommer inte att avkodas och visas alls. Mest troligt kommer detta att få videon att “frysa” för ett ögonblick.

För att undvika (åtminstone försöka undvika) att videon fryser eller är full av avkodningsartefakter använder WebRTC negativa bekräftelsemeddelanden (NACK). När mottagaren inte får ett förväntat RTP-paket returnerar det ett NACK-meddelande för att be avsändaren att skicka det saknade paketet igen. Mottagaren *väntar* på omskickandet av paketet. Sådana omskickningar orsakar givetvis ökad latens. Antalet skickade och mottagna NACK-paket registreras i WebRTCs inbyggda statistikfält outbound stream nackCount och inbound stream nackCount.

Du kan se fina diagram över inkommande och utgående `nackCount` på webrtc internals sidan. Om du ser att `nackCount` ökar, betyder det att nätverket upplever höga paketförluster, och WebRTC-stacken gör sitt bästa för att skapa en smidig video och ljudupplevelse trots det.

När paketförlusten är så hög att avkodaren inte kan producera en bild eller efterföljande beroende bilder, som i fallet med en helt förlorad I-bild, kommer alla framtida P-bilder inte att avkodas. Mottagaren kommer att försöka mildra det genom att skicka ett speciellt bildförlustindikationsmeddelande (Picture Loss Indication message, PLI). När sändaren får ett `PLI` meddelande kommer den

att skicka en ny I-bild för att hjälpa mottagarens avkodare. I-bilder är normalt större än P-bilder, så det ökar mängden data som behöver sändas. Precis som med NACK-meddelanden måste mottagaren vänta på den nya I-bilden vilket ökar latensen ytterligare.

Kolla efter `pliCount` på webrtc internals sidan. Om den ökar, justera din kodare så att den skickar mindre data, eller producerar paket med mer redundans.

Latens på mottagarsidan Latens kommer att påverkas av paket som kommer fram i fel ordning. Om den nedre halvan av en bild kommer före toppen måste du vänta på toppen innan du kan avkoda den. Detta förklaras mer noggrant i kapitlet Solving Jitter.

Du kan också hänvisa till det inbyggda jitterBufferDelay mätningen för att se hur länge en bild hölls i mottagningsbufferten i väntan på alla paket, innan den skickades vidare till avkodaren.

Historia

Detta kapitel skrivs fortfarande och vi har inte alla fakta än. Vi genomför intervjuer för att bygga en mer samlad historia av digital kommunikation.

RTP

RTP och RTCP är protokollet som hanterar all mediatransport för WebRTC. Det definierades i RFC 1889 i januari 1996. Vi är väldigt lyckliga att få en av författarna Ron Frederick prata om det själv. Ron laddade nyligen upp Network Video tool på GitHub, ett projekt som la grunden till RTP.

I hans egna ord

(Originaltext på engelska)

I oktober 1992 började jag experimentera med Sun VideoPix frame grabber med tanken att skriva ett nätverksvideokonferensverktyg baserat på IP multicast. Det modellerades efter “vat” - ett telefonkonferensverktyg utvecklat på LBL, eftersom det använde ett liknande enkelt protokoll där användare som vill gå med i konferenser helt enkelt bara skickade data till en viss multicast-grupp och samtidigt lyssnade på den gruppen för ta emot video från andra medlemmar.

För att programmet verkligen skulle lyckas behövde det komprimera videodata innan den skickas till nätverket. Mitt mål var att göra en tillräkligt bra videoström som skulle rymmas i cirka 128 kbps, eller en den normala bandbredden tillgänglig på en vanlig ISDN-modem linje. Jag hoppades också att göra något som fortfarande såg ok ut men bara använde halva denna bandbredd. Detta betydde att jag behövde ungefär en faktor 20 i kompression för den specifika bildstorlek och bildhastighet jag arbetade med. Jag kunde uppnå denna komprimering och

ansökte om patent på de tekniker som jag använde, senare beviljat i patent US5485212A: Videokomprimering av programvara för telekonferenser.

I början av november 1992 släppte jag videokonferensverktyget "nv" (själva programmet) gratis på nätet. Efter några initiala tester användes programmet för att sända delar av November Internet Engineering Task Force till hela världen. Cirka 200 subnät i 15 länder kunde ta emot denna sändning, och cirka 50-100 personer fick video med "nv" någon gång under veckan.

Under de kommande månaderna sändes tre andra workshops och några mindre möten via "nv" till internet, inklusive Australiska NetWorkshop, MCNC Packet Audio and Video workshoppen och MultiG-workshoppen om distribuerad virtuell verklighet från Sverige.

En release av källkoden för "nv" följde i februari 1993 och i mars släppte jag en version av verktyget där jag introducerade ett nytt wavelet-baserat komprimeringsschema. I maj 1993 lade jag till stöd för färgvideo.

Nätverksprotokollet som används för "nv" och andra internetkonferensverktyg blev grunden för Realtime Transport Protocol (RTP). Det standardiserades genom Internet Engineering Task Force (IETF), först publicerat i RFC 1889 - 1890 och senare reviderad i RFC 3550 - 3551 tillsammans med olika andra RFC:er som täcker profiler för att skicka specifika format av ljud och video.

Under de följande åren fortsatte arbetet med "nv" och verktyget portades till ytterligare ett antal hårdvaruplattformar och videoinspelare. Det fortsatte att användas som ett av de främsta verktygen för sändning konferenser på internet vid den tiden, inklusive av NASA för att sända live-video av uppskjutningar av rymdskytteln på nätet.

1994 lade jag till stöd i "nv" för videokomprimeringsalgoritmer utvecklat av andra, inklusive vissa hårdvarukomprimeringsscheman som CellB-format som stöds av SunVideo-videoinspelningskortet. Det gjorde det möjligt för "nv" att skicka video i CUSeeMe-format, för att skicka video till användare som körde CUSeeMe på Mac och PC.

Den senast släppta versionen av "nv" var version 3.3beta, släppt i juli 1994. Jag arbetade med en "4.0alpha"-release som var avsedd att migrera "nv" till version 2 av RTP-protokollet, men detta arbete blev aldrig slutfört på grund av att jag gick vidare till andra projekt. En kopia av 4.0alfa ingår i Network Video tool arkivet, men det är oavslutat och det finns kända problem med det, särskilt i det ofullständiga RTPv2-stödet.

Ramverket i "nv" blev senare grunden för video konferenser i projektet "Jupiter multimedia MOO" på Xerox PARC, som så småningom blev grunden för ett avknoppningsföretag "PlaceWare", senare förvärvades av Microsoft. Det användes också som grund för ett antal hårdvarukonferensprojekt som tillät sändning av fullständig NTSC video över Ethernet och ATM-nätverk med hög bandbredd. Jag använde också senare en del av den här koden som grund för "Mediastore", som var en nätverksbaserad videoinspelnings- och uppspelningstjänst.

Kommer du ihåg de andra personernas motiv/idéer i utkastet?

Vi var alla forskare som arbetade med IP-multicast och hjälpte till att skapa internet-multicast-systemet (även kallad MBONE). MBONE skapades av Steve Deering (som först utvecklade IP-multicast), Van Jacobson och Steve Casner. Steve Deering och jag hade samma rådgivare på Stanford och Steve slutade arbeta på Xerox PARC när han lämnade Stanford. Jag tillbringade en sommar på Xerox PARC som praktikant för att arbeta med IP multicast-relaterade projekt och fortsatte att arbeta för dem på deltid, och senare på full tid medan jag var på Stanford. Van Jacobson och Steve Casner var två av de fyra författarna på de första RTP-RFC:erna, tillsammans med Henning Schulzrinne och jag själv. Vi alla hade MBONE-verktyg som vi arbetade med som möjliggjorde olika former av online-samarbete och försöker komma på ett gemensamt basprotokoll alla dessa verktyg som kunde användas var det som ledde till RTP.

Multicast är superfascinerande. WebRTC är enkelsändning, kan du utveckla det lite mer?

Innan jag kom till Stanford och lärde mig om IP-multicast hade jag en lång historia av att arbeta med olika sätt för människor att använda datorer som ett sätt kommunicera med varandra. Detta började i början av 80-talet för mig när jag körde en digital anslagstavla (BBS) där människor kunde logga in och lämna meddelanden till varandra, båda privata (typ av motsvarande e-post) och offentliga (diskussionsgrupper). Ungefär under samma tid lärde jag mig också om onlinetjänstleverantören CompuServe. En av de coola funktionerna på CompuServe var något som kallades "CB Simulator" där människor kunde prata med varandra i realtid. Det var helt textbaserat, men det hade en koncept med "kanaler" som en riktig CB-radio och flera personer kunde se vad andra skrev, så länge de var i samma kanal. Jag byggde min egen version av CB som kördes på ett timesharing-system jag hade åtkomst som lät användare på systemet skicka meddelanden till varandra i realtid, och under de närmaste åren arbetade jag med några vänner med att utveckla mer sofistikerade versioner av kommunikationsverktyg i realtid på flera olika datorsystem och nätverk. I själva verket en av dessa system är fortfarande i drift, och jag använder det varje dag för att prata med folk jag gick till college med för 30+ år sedan!

Alla dessa verktyg var textbaserade, eftersom datorer på den tiden oftast saknade ljud- och videofunktioner, men när jag kom till Stanford och lärde mig om IP-multicast blev jag fascinerad av tanken att använda multicast för att bygga något mer som en riktig "radio", där du kan skicka en signal ut på nätverket som inte riktades till en enskild användare, men istället kan alla som ställde in den "kanalen" ta emot den. Av en händelse så var den dator jag portade IP-multicast-koden till av den första generationen SPARC-station från Sun, och den hade faktiskt inbyggt ljud av telefonkvalitet i hårdvaran! Du kan digitalisera ljud från en mikrofon och spela upp det igen på inbyggda högtalare (eller via ett headset). Så min första tanke var att ta reda på hur man skickar ljud över nätverket i realtid med IP-multicast, och se om jag kunde bygga en motsvarande

“CB-radio”, men med riktigt ljud istället för text.

Det fanns några knepiga problem att lösa, som det faktum att datorn kunde bara spela en ljudström åt gången, så om flera personer pratade du behövde man matematiskt “blanda” flera ljudströmmar till en enda så att du kunde spela upp det, men det kunde göras i mjukvara när man förstått hur ljud-samplingen fungerade. Den ljudapplikationen ledde mig till att arbeta med MBONE och så småningom gå från ljud till video med “nv”.

###Har något som utelämnats från protokollet som du önskar att du hade lagt till? Är det något i protokollet du ångrar?

Jag skulle inte säga att jag ångrar det, men ett vanligt klagomål som människor har om RTP är komplexiteten i att implementera RTCP, kontrollprotokollet som körs parallellt med RTP-datatrafiken. Jag tror att komplexiteten var en stor del av varför RTP inte antogs i större utsträckning, särskilt i enkelsändningsfallet där det inte fanns lika stort behov av RTCPs funktioner. När nätverksbandbredden blev mindre begränsad och trängseln inte var ett lika stort problem, började många människor helt enkelt streama ljud och video över vanlig TCP (och senare HTTP), och i allmänhet fungerade det “tillräckligt bra”, så det var inte värt att använda RTP.

Tyvärr innebär användning av TCP eller HTTP att ljud och video från flera applikationer måste skicka samma data över nätverket flera gånger, till var och en av alla klienter som vill ta emot det, vilket gör det mycket mindre effektivt ur ett bandbreddsperspektiv. Jag önskar ibland att vi hade drivit på hårdare att få IP-multicast accepterat utanför forskarkretsarna. Jag tror att vi kunde ha sett övergången från kabel och analog TV till internetbaserad ljud och video mycket tidigare om vi hade gjort det.

Vilka saker trodde du skulle byggas med RTP? Har något coolt RTP

projekt eller idé som aldrig blev av?

En rolig sak jag byggde var en version av det klassiska ”Spacewar”-spelet som använde IP-multicast. Utan att ha någon form av central server kunde flera klienter var och en köra spacewar programmet och börja sända sina skepps plats, hastighet, riktning, och liknande information för eventuella ”kulor” som den avfyrat, och alla andra instanser av programmet plockade upp den informationen och använde den lokalt, så att användarna kan alla se varandras skepp och kulor, med skepp som “exploderar” om de kraschar in i varandra eller om kulorna träffar dem. Jag gjorde till och med ”skräp” från explosioner till levande föremål som kan ta ut andra skepp, ibland ledande till roliga kedjereaktioner!

I andan i det ursprungliga spelet byggde jag det med simulerad vektor grafik, så du kan göra saker som att zooma in och ut och se all grafik skala upp och ner. Fartygen själva var en massa linjesegment i vektorform som jag fick några av mina kollegor på PARC att hjälpte mig att designa, så alla spelares skepp hade ett unikt utseende.

I grund och botten kan alla program som kan dra nytta av en realtidsdataström som inte kräver perfekt leverans av paket i ordning dra nytta av RTP. Så förutom ljud och video kunde vi bygga saker som en delad whiteboard. Till och med fil-överföringar kan dra nytta av RTP, särskilt i samband med IP-multicast.

Tänk dig något som BitTorrent, men där du inte behöver skicka all daata från punkt till punkt mellan klienterna. Den ursprungliga källan kunde skicka en multicast-ström till alla nerladdare på en gång och eventuella paketförluster på vägen kunde snabbt rensas upp genom en återöverföring från andra klienter som redan fått den datan. Du kan till och med begränsa din återutsändningsförfrågan så att någon peer i närheten skickar kopian av datan, och även den sändningen kan vara multicast för andra i den regionen, eftersom en paketförlust i mitt i nätverket ofta betyder at en massa klienter nedströms från den punkten alla saknar samma data.

Varför var du tvungen att bygga din egen videokomprimering. Fanns det inget annat tillgängligt då?

När jag började bygga "nv" använde de enda systemen jag kände till för videokonferenser mycket dyr och specialiserad hårdvara. Till exempel, Steve Casner hade tillgång till ett system från BBN som kallades "DVC" (och senare kommersialiserad som "PictureWindow"). Komprimering krävde specialiserad hårdvara, men dekompressionen kunde göras i programvara. Det som gjorde "nv" unikt var att både kompression och dekompression gjordes i programvara, och det enda hårdvarukravet var att ha något för att digitalisera en inkommande analog videosignal.

Många av de grundläggande begreppen om hur man komprimerar video fanns redan då, till exempel dök MPEG-1-standarden upp precis vid samma tid som "nv", men kodning i realtid med MPEG-1 var definitivt INTE möjligt på den tiden. De förändringar jag gjorde handlade om att ta dessa grundläggande koncept men använda mig av mycket effektivare algoritmer där jag undvek saker som cosinus transformerar och flyttalsberäkningar, och till och med undvvek heltalsmultiplikationer eftersom de var mycket långsamma på SPARC-stations. Jag försökte göra så mycket som möjligt med bara addition, subtraktion, bitmasker, och bitskiftning, och det blev tillräckligt snabbt för att fortfarande kännas som en video.

Inom ett år eller två från släppet av "nv" fanns det många olika ljud och videoverktyg att välja mellan, inte bara på MBONE utan på andra platser som CU-SeeMe-verktyget byggt på Mac. Så det var helt klart en idé vars tid hade kommit. Jag gjorde faktiskt "nv" kompatibelt med många av dessa verktyg, och i ett fåtal fall plockade andra verktyg upp min "nv"-codec, så att de kunde kommunicera via min komprimeringsalgoritm.

WebRTC

WebRTC krävde ett större standardiseringsarbete än alla andra insatser som beskrivs i detta kapitel. Det krävde samarbete mellan två olika standardorgan (IETF och W3C) och hundratals individer i många företag och länder. För att ge oss en bild av motivationen och den monumentala ansträngning som krävdes för att få WebRTC att hända har vi frågat Serge Lachapelle.

Serge jobbar på Google, för närvarande produktchef för Google Workspace. Det här är min sammanfattning av intervjun.

Vad fick dig att börja jobba på WebRTC?

Jag har varit intresserad av att bygga kommunikationsverktyg ända sedan jag var i college. På 90-talet började teknik som nv dyka up, men den var svår att använda. Jag skapade ett projekt som användarna gå med i ett videosamtal direkt från webbläsaren. Jag portade det också till Windows.

Jag tog med den erfarenheten till Marratech, ett företag som jag grundade. Vi skapade programvara för gruppvideokonferenser. Tekniskt sett var landskapet helt annorlunda då. Lösningar i framkanten för video baserades på multicast-nätverk. En användare var beroende av nätverket för att leverera till ett videopaket till alla andra i samtalet. Detta innebar att vi hade mycket enkla servrar. Men det har också en stor nackdel, nätverket måste konfigureras för att stödja det. Branschen slutade använda multicast och gick över till paketblandare (packet shufflers), mer allmänt känt som SFU.

Marratech förvärvades av Google 2007. Jag fortsatte att arbeta med projektet som var föregångaren till WebRTC.

Det första Google-projektet

Det första projektet som det framtida WebRTC-teamet arbetade med var Gmail ljud och videochatt. Att få in ljud och video i webbläsaren var ingen lätt uppgift. Det krävde specialkomponenter som vi var tvungna att licensiera från olika företag. Ljud licensierades från GIP, video licensierades för Vidyo och nätverket var libjingle. Magin var att få alla att fungera tillsammans.

Varje delsystem har helt olika API:er och antog att du löste olika problem. För att allt ska fungera tillsammans behöver du kunskap om nätverk, kryptografi, media med mera. Justin Uberti var den som tog sig an detta arbete. Han kopplade ihop komponenterna till en användbar produkt.

Att rendera i realtid i webbläsaren var också riktigt svårt. Vi var tvungna att använda NPAPI (Netscape Plugin API) och göra massor av smarta saker för att få det att fungera. De lärdomar vi fick från det projektet påverkade WebRTC mycket.

Chrome

Samtidigt startade Chrome-projektet hos Google. Det var så mycket entusiasm, och detta projekt hade stora planer. Det pratades om WebGL, offline-stöd, databasfunktioner, låg latenskontroll för spel bara för att nämna några.

Att flytta bort från NPAPI blev ett stort fokus. Det är ett kraftfullt API, men kommer med stora säkerhetsmässiga konsekvenser. Chrome använder en sandlåda för att skydda användarna. Operationer som kan vara potentiellt osäkra körs i olika processer. Även om något går fel har en angripare fortfarande inte tillgång till användardatan.

Hur WebRTC kom till

För mig skapades WebRTC utifrån några specifika motiv som tillsammans ledde fram till projektet.

Det borde inte vara så svårt att bygga RTC-upplevelser. Så mycket ansträngning går åt till att återimplementera samma sak. Vi borde lösa dessa frustrerande integrationsproblem en gång för alla, och sen fokusera på andra saker.

Mänsklig kommunikation bör vara oförhindrad och öppen. Varför är det ok att text och HTML är öppet, men min röst och video i realtid är inte det?

Säkerhet är en prioritet. Att använda NPAPI var inte bäst för användarna. Detta var också en chans att skapa ett protokoll som är säkert från start.

För att få WebRTC att bli av köpte Google de komponenter vi hade använts tidigare och släppte källkoden till dem. On2 var förvärvat för sin videoteknik och Global IP Solutions för sin RTC-teknik. Jag hade ansvaret för att förvärva GIPS. Vi fick arbeta med att kombinera dessa och göra dem lätta att använda inom och utanför webbläsaren.

Standardisering

Standardisering av WebRTC var något vi verkligen ville göra, men inte något jag eller någon i vårt team hade gjort tidigare. Som tur var hade vi Harald Alvestrand på Google. Han hade gjort omfattande arbete i IETF redan och startade WebRTC-standardiseringsprocessen.

Sommaren 2010 planerades en informell lunch i Maastricht. Utvecklare från många företag kom tillsammans för att diskutera vad WebRTC borde vara. Lunchen hade ingenjörer från Google, Cisco, Ericsson, Skype, Mozilla, Linden Labs med flera. En lista av alla närvarande och själva presentationen finns på rtc-web.alvestrand.com.

Skype gav också bra vägledning på grund av det arbete de hade gjort med Opus i IETF.

Att stå på jättarnas axlar

När du arbetar i IETF bygger du vidare det arbete som gjorts före dig. Med WebRTC hade vi turen att många saker redan fanns. Vi behövde inte ta oss an alla problem eftersom många redan var lösta. Om du inte gillar den redan befintliga tekniken kan det dock vara frustrerande. Det måste vara en ganska stark anledning att bortse från befintligt arbete, så att skapa en egen är sällan ett alternativ.

Vi valde medvetet inte att standardisera saker som signalering. Detta hade redan lösts med SIP och andra ansträngningar utanför IETF, och det kändes som att det kunde bli mycket politiska debatter. Slutsatsen var att det inte kändes some det fanns mycket att lägga till på det området.

Jag var inte lika involverad i standardiseringen som Justin och Harald, men jag tyckte det var kul att göra. Jag var mer intresserat att återvända till att bygga lösningar för användare.

Framtiden

WebRTC är i ett bra läge idag. Det finns många inkrementella förändringar under utveckling, men inget speciellt som jag har jobbat med.

Jag är mest intresserad att se vad molnet kan göra för kommunikation. Genom att använda avancerade algoritmer kan vi ta bort bakgrundsbrus från ett samtal och göra kommunikation möjlig där det inte var möjligt tidigare. Vi ser också att WebRTC sträcker sig långt bortom kommunikation... Vem kunde tro att det skulle driva moln-baserade spel 9 år senare? Allt detta skulle vara omöjligt utan grunden från WebRTC.

FAQ

{{<details “Varför använder WebRTC UDP?”>}} NAT Traversal kräver UDP. Utan att NAT Traversal är det inte möjligt att sätta upp en P2P-anslutning. UDP har inte “garanterad leverans” som TCP, så WebRTC sköter det på applikationsnivån istället.

Se [Connecting]({{< ref “03-connecting” >}}) för mer info. {{

}}

{{<details “Hut många DataChannels kan jag ha?”>}} 65534 kanaler, eftersom strömidentifieraren har 16 bitar. Du kan när som helst stänga eller öppna nya kanaler. {{

}}

{{<details “Begränsar WebRTC bandbreden på något sätt?”>}} Både DataChannels och RTP använder trängselskontroll. Detta innebär att WebRTC aktivt mäter din bandbredd och försöker använda den optimala mängden. Det

är en balansgång att skicka så mycket data som möjligt utan att överväldiga anslutningen. {{

}}

{{<details “Kan jag skicka binär data?”>}} Ja, du kan skicka både text och binär data över DataChannels. {{

}}

{{<details “Vilken latens kan jag förvänta mig från WebRTC?”>}} För icke-optimerad media kan du förvänta dig under 500 millisekunder. Om du tar dig tid att optimera videon eller kan tumma lite på kvalitèten och istället optimera för latens, har utvecklare fått ner den under 100 ms.

DataChannels har alternativet “partiell tillförlitlighet”, vilket kan minska latens orsakad av tappade paket. Om det har konfigurerats korrekt har det visat sig vara snabbare än TCP TLS-anslutningar. {{

}}

{{<details “Varför skulle jag vilja ignorera paketordningen i DataChannels?”>}} När nyare information gör den gamla värdelös, till exempel positionsinformation för ett objekt, eller när varje meddelande är helt oberoende av de andra och vi vill undvika head-of-line blockering. {{

}}

{{<details “Kan jag skicka ljud eller video över en DataChannel?”>}} Ja, du kan skicka vilken data som helst via en DataChannel. I webbläsarfallet är det då ditt ansvar att avkoda data och skicka dem till en mediaspelare för uppspelning, medans allt det görs automatiskt om du använder mediekanaler. {{

}}

Ordlista

- ACK: Kort för acknowledgment, bekräftelse
- AVP: Audio and Video profile
- B-Frame: Bi-directional Predicted Frame. En partiell bildruta, en modifiering av föregående och nästkommande bilder.
- DCEP: Data Channel Establishment Protocol definierat i RFC 8832
- DeMux: Demultiplexer
- DLSR: Delay since last sender report, fördröjningstid
- DTLS: Datagram Transport Layer Security definierat i RFC 6347
- E2E: End-to-End, dvs hela vägen
- FEC: Forward Error Correction
- FIR: Full INTRA-frame Request
- G.711: En smalbandig ljud-kodek
- H.264: Avancerad video-kodek

- H.265: Konkret specifikation för ITU-T H.265 video kodeken
- HEVC: High Efficiency Video Coding
- HTTP: Hypertext Transfer Protocol
- HTTPS: HTTP över TLS, definierat i RFC 2818
- I-Frame: Intra-coded Frame. En komplett bild, kan avkodas utan extra information.
- ICE: Interactive Connectivity Establishment definierat i RFC 8445
- INIT: Initiering
- IoT: Internet of Things, små uppkopplade enheter
- IPv4: Internet Protocol, Version 4
- IPv6: Internet Protocol, Version 6
- ITU-T: International Telecommunication Union Telecommunication Standardization Sector
- JSEP: JavaScript Session Establishment Protocol definierat i RFC 8829
- MCU: Multi-point Conferencing Unit
- mDNS: Multicast DNS definierat i RFC 6762
- MITM: Man-In-The-Middle, avlyssning av trafik
- MTU: Maximum Transmission Unit, paket storleken
- MUX: Multiplexing
- NACK: Kort för Negative Acknowledgment, negativ bekräftelse
- NAT: Network Address Translation definierat i RFC 4787
- Opus: En helt öppen, royalty-fri och väldigt anpassningsbar ljud-kodek
- P-Frame: Predicted Frame. En partiell bild som bara innehåller skillnaden från den föregående bilden.
- P2P: Peer-to-Peer, kommunikation direkt mellan två klienter
- PLI: Picture Loss Indication
- PPID: Payload Protocol Identifier
- REMB: Receiver Estimated Maximum Bitrate, ett sätt att beräkna bandbredd
- RFC: Request for Comments
- RMCAT: RTP Media Congestion Avoidance Techniques
- RR: Receiver Report
- RTCP: RTP Control Protocol definierat i RFC 3550
- RTP: Real-time transport protocol definierat i RFC 3550
- RTT: Round-Trip Time
- SACK: Selective Acknowledgment
- SCTP: Stream Control Transmission Protocol definierat i RFC 4960
- SDP: Session Description Protocol definierat i RFC 8866
- SFU: Selective Forwarding Unit
- SR: Sender Report, avsändarens rapport
- SRTP: Secure Real-time Transport Protocol definierat i RFC 3711
- SSRC: Synchronization Source
- STUN: Session Traversal Utilities for NAT definierat i RFC 8489
- TCP: Transmission Control Protocol, protokoll med leveransgaranti
- TLS: Transport Layer Security, definierat i RFC 8446
- TMMBN: Temporary Maximum Media Stream Bit Rate Notification

- TMMBR: Temporary Maximum Media Stream Bit Rate Request
- TSN: Transmission Sequence Number
- TURN: Traversal Using Relays around NAT defined in RFC 8656
- TWCC: Transport Wide Congestion Control
- UDP: User Datagram Protocol, protokoll utan leveransgaranti
- VP8, VP9: Mycket effektiva videokompressionsteknologier (video "kodeks") utvecklade av WebM Projektet. Fria att använda utan att betala royalty.
- WebM: Ett öppet media-filformat utvecklat för webben.
- WebRTC: Web Real-Time Communications. W3C WebRTC 1.0: Real-Time Communication Between Browsers