# Feladatkiírás

A feladat egy WebRTC technológiára épülő, böngészőben futó csevegő alkalmazás elkészítése. A megoldásnak támogatnia kell a szöveges, videó és hang alapú csevegést is, a kliensek közötti (peer-to-peer) kapcsolatok segítségével. Az elkészült alkalmazásban, egy felhasználó kiválaszthat egy csoportot a többi online felhasználó közül, akikkel csevegni szeretne. Ekkor a kiválasztott módszertől függően, vagy csoportos üzenetet küld nekik, vagy csoportos hívást kezdeményez, esetleg videó konferenciát indít. A megoldás elkészítéséhez szükséges bizonyos központi architektúrális komponensek implementálása is, de a feladat fókusza a kliensek közötti kommunikáción van. A hallgató feladatának a következőkre kell kiterjednie:

* Ismerje meg és mutassa be a WebRTC szabvány fő komponenseit, azok funkcióit!
* Elemezze a különböző hálózati infrastruktúra elemek (tűzfalak, NAT) hatását a WebRTC-s peer-to-peer kapcsolatokra.
* Készítse el és tesztelje a rendszer implementációját a három (szöveg, hang, videó) adat típus átvitelére.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

# Fedlap

Juhász Bálint

WebRTC alapú csevegő alkalmazás, hang és videó támogatással

**Konzulens**

Jánoky László Viktor

Budapest, 2018

# Tartalomjegyzék

Contents

[Feladatkiírás 1](#_Toc530908458)

[Fedlap 2](#_Toc530908459)

[Tartalomjegyzék 3](#_Toc530908460)

[Hallgatói Nyilatkozat 5](#_Toc530908461)

[Összefoglaló 6](#_Toc530908462)

[Abstract 8](#_Toc530908463)

[Szakterület és probléma bemutatása 9](#_Toc530908464)

[Valós idejű kommunikáció 9](#_Toc530908465)

[Bevezető 9](#_Toc530908466)

[Network address translation (NAT) 9](#_Toc530908467)

[Session Traversal Utilities for NAT (STUN) 13](#_Toc530908468)

[Traversal Using Relays around NAT (TURN) 13](#_Toc530908469)

[Interactive Connectivity Establishment (ICE) 14](#_Toc530908470)

[Session Description Protocol (SDP) 16](#_Toc530908471)

[WebRTC 21](#_Toc530908472)

[Lehetőségek 24](#_Toc530908473)

[WebRTC 24](#_Toc530908474)

[Egy-egy 24](#_Toc530908475)

[Egy-több 24](#_Toc530908476)

[Több-több 24](#_Toc530908477)

[Létező megoldások 26](#_Toc530908478)

[Elkészült rendszer bemutatása 26](#_Toc530908479)

[Célkitűzések 26](#_Toc530908480)

[Tervezési szempontok 26](#_Toc530908481)

[Egy-egy kapcsolatú demó 26](#_Toc530908482)

[Megvalósítás 26](#_Toc530908483)

[WebRTC könyvtár 29](#_Toc530908484)

[Fájl küldés 30](#_Toc530908485)

[UI 30](#_Toc530908486)

[Összefoglaló 33](#_Toc530908487)

[Több-több kapcsolatú webapp 33](#_Toc530908488)

[Bevezető 33](#_Toc530908489)

[Első verzió 34](#_Toc530908490)

[Második verzió 37](#_Toc530908491)

[Elkészült rendszer értékelése 38](#_Toc530908492)

[Ellenőrzés 38](#_Toc530908493)

[Teljesítmény 38](#_Toc530908494)

[Szerzett tapasztalatok 38](#_Toc530908495)

[Összefoglaló 38](#_Toc530908496)

[Függelék 38](#_Toc530908497)

# Hallgatói Nyilatkozat

Alulírott Juhász Bálint, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot/ diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2018. 05. 12.

........................................................

Juhász Bálint

# Összefoglaló

Az idő előrehaladtával egyre több funkciót támogatnak a böngészők. Ezek közé tartozik a WebRTC is, ami peer to peer hálózat kialakítását teszi lehetővé. De mi is az a WebRTC? Az RTC feloldása Real Time Communication, tehát valós idejű kommunikációt valósít meg. Ilyen megoldás a VOIP, illetve a WebRTC is. A kommunikáció UDP fölött zajlik, mivel a TCP nem rendelkezik a stream-eléshez alkalmas latency-vel. A WebRTC, ugyanúgy ahogy a VOIP is, SDP header megosztásával épít fel pont-pont kapcsolatot full duplex módon. Amennyiben mindkét kliens elérhető egymás felé saját ip címükön, a kapcsolat egyszerű SDP cserével épül fel, míg NAT hálózatokon át szükség van egymás publikus ip címére ugyanehhez. Ha ismerjük a saját publikus ip címünket, még mindig nincs gond, viszont ha nem, már szükség van egy publikus címen elérhető szerverre, ami meg tudja osztani a kliensekkel, hogy ők milyen publikus címen érhetőek el. Emiatt sajnos a WebRTC a gyakorlatban nem tekinthető teljesen decentralizáltnak, csak ha minden kliensnek egyedi címe van és ugyanazon a hálózaton vannak.

A WebRTC nem csak a böngésző forráskódjában, de API szinten is elég komplex, annak ellenére, hogy mivel kliens oldal, ezért JavaScriptben általában próbálnak minden API-t egyszerűvé tenni. Emiatt az első feladatom az volt, hogy írjak egy demó oldalt, ahol szerverek nélkül fel lehet építeni kapcsolatot két gép között. Ehhez egy egyszerűbb JS library-t is kellett írnom a WebRTC köré. Az SDP header-ök megosztását a kliensek Miután ezt sikerült elérnem, áttértem a fő feladatomra, egy Skype jellegű alkalmazásra, ahol több ember beszélgethet egymással minden kényelmetlenség nélkül. Ehhez már Signaling Server kell, ami az SDP header cserét elvégzi. Ahhoz, hogy a NAT mögötti gépek is kapcsolódni tudjanak egymáshoz, STUN vagy TURN szerverre van szükség. A STUN csak publikus ip-t szolgáltat vissza, a TURN pedig a kapcsolatfelvételtől kezdve minden üzenetet tükröz a másik félnek.

A több kliens közötti kommunikáció felvet egy toávbbi problémát. Ha N ember szeretne egymással beszélni, az egy böngészőre vetítve 2N darab stream, ami nem működőképes módszer. Emiatt médiaszervereket kell alkalmazni, amik ezt a számot limitálják. Amennyiben kapcsolatonként szeretnénk közbeavatkozni egy szerverrel, egy kliensnek elég N+1 stream, ha pedig egy eggyé renderelt képet szeretnénk kapni a többi klienstől, elég 2 csatorna, egy fel és egy le. Természetesen a szerver az utóbbi esetben sokkal több számítást végez.

Mint látható, egy WebRTC szolgáltatás létrehozása nagyon sok szinten kíván elmélyült tudást, főleg azért, mert akár 5 különböző szervernek is részt kell vennie, hogy gondtalanul működjön a szolgáltatás.

# Abstract

# Szakterület és probléma bemutatása

Ebben a fejezetben a technológia elméleti részét, megfelelő előzetes ismereteket, illetve a WebRTC technológia működését mutatom be.

## Valós idejű kommunikáció

### Bevezető

A valós idejű kommunikáció sok használati módot fed le. Mi most főként a media streaming-gel fogunk foglalkozni. Ha van valamilyen tartalmunk, legyen az videó, audió, nyers adat, különböző elvárásoknak kell megfelelnie a csatornának. Videó és audió esetén ez a késleltetés minimalizálása és a minőség megtartása, míg adat csatorna esetén az üzenet változatlansága. Ideális esetben az átviteli közeg ezt garantálja, hiszen végtelen sok adatot tud szállítani késleltetés nélkül. Ezzel szemben a valóságban mindenhol kompromisszumokat kell kötni mely hol a késleltetés, hol a minőség, hol a hibátlan átvitel rovására megy. Egy ilyen környezetben kell használható megoldást találni a stream-elésre. Ez a valós idejő kommunikáció problémaköre.

Mielőtt rátérnék a WebRTC által alkalmazott legfontosabb protokollra, meg kell említenem néhány technológiát, melyre az támaszkodik.

### Network address translation (NAT)

Elsőként a NAT-ot mutatom be. A WebRTC-nek ez a technológia okoz a legnagyobb gondot. Bár a probléma megoldására születtek a WebRTC-t segítő protokollok, hogy teljes képet kaphassunk a problémáról, e fejezet részletesen bemutatja a NAT technikákat.

<https://en.wikipedia.org/wiki/Network_address_translation>  
https://www.think-like-a-computer.com/2011/09/16/types-of-nat/  
https://tools.ietf.org/id/draft-takeda-symmetric-nat-traversal-00.txt

A NAT-ot eredetileg ip címek egy-egy megfeleltetésű fordítására használták. Ha például egy iroda költözött, a gépek megtarthatták a címüket, a kimenő/bemenő kéréseket a router átfordította. A lokális címet a kiosztott publikus címre, illetve fordítva. Ezt hívják **Basic NAT**-nak.

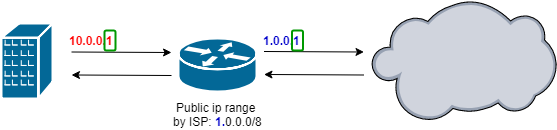


Figure 1. Basic NAT a gyakorlatban

Valójában a NAT-ot sok lokális ip kevés (legtöbbször 1) publikus ip címre való leképezésére használják. Ezt hívják **One-to-many NAT**-nak. Ilyenkor nincs kölcsönös megfeleltetés a két tartomány között, hanem a routernek más helyen kell eltárolnia azt az információt, hogy melyik lokális címről érkezett a kérés. TCP és UDP esetén ez a meta-adat a portszámban foglal helyet. Ha egy kliens a routeren keresztül el akar érni egy erőforrást, a router egy még nem foglalt source portszámhoz köti a klienst és erre a portra várja vissza is a választ. Így sikeresen vissza tud jutni a válasz a klienshez. *(megjegyzés: mivel a portszám 16 biten tárolódik a TCP és UDP header-ben, hiába van /8-as subnete egy hálózatnak, csak 65536 kliens kommunikálhat egyszerre kifelé, a maradék 16 millió addig nem éri el az internetet)*

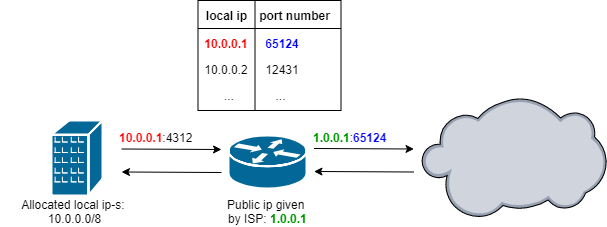


Figure 2. Many-to-one a gyakorlatban

Láthatjuk, hogy a One-to-many NAT-nak sokféle megoldása lehet, ha figyelembe vesszük, hogy nem csak request-response viselkedéső forgalom lehet a routeren keresztül, de például egy egy kívülről jövő kérést is le kell kezelni. A következő fordítási módszerek a legelterjedtebbek:

**Full-cone NAT (static NAT):** A basic NAT megfeleltetése úgy, hogy mivel csak egy publikus ip cím van, a lokális címeket portként azonosítja. Előnye, hogy amíg a kliens elérhető a belső hálózaton, kívülről bárki kommunikálhat vele. Hátránya, hogy csak egy porton keresztül lehet kommunikálni vele. *(megjegyzés: amikor kézzel állítunk be egy ilyen NAT-olást néhány gépre, azt hívják port forwardingnak. Így elérhetővé tehetünk lokális szervereket az interneten, azzal a hátránnyal, hogy egy portot csak az egyikük használhatja)*

*[Lásd:* Basic NAT a gyakorlatban*]*

**Address-restricted cone NAT:** Ugyanaz a működési elv mint a Full-cone-nál, viszont a NAT publikus oldalán lévő kliens csak akkor kapcsolódhat a lokális hálózaton lévő kliensre, ha az már forgalmazott felé. Viszont amennyiben már szabad kérést intéznie felé, a külső kliens bármely portjáról kapcsolódhat rá.

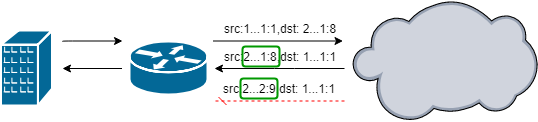


Figure 3. Az ARC NAT nem engedi be a kliensre a 2.0.0.2 címről érkező kérést

**Port-restricted cone NAT:** Ugyanaz a működési elv mint az address-restricted változatnál, de itt a külső kliens csak azon a porton indíthat kérést a lokális felé, amivel előtte a lokális kliens küldött adatot felé.

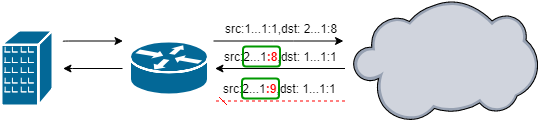


Figure 4. Azonos címről sem enged be, ha a port szám nem egyezik meg azzal, amire a lokális kliens kiküldte az adatot.

**Symmetric NAT:** Ugyanaz a működési elv mint a post restricted NAT-nál, viszont minden egyes TCP vagy UDP kapcsolat a lokális kliens felől a routeren külön portról indul, így csak az a publikus kliens küldhet vissza adatot, amelyiknek a felkérést címezték.

Fontos megemlítenem a **hole punching**-ot, mint eljárást. Lényege, hogy egy publikus szerver segítségével a két NAT mögötti kliens egymás felé tud üzentet küldeni aminek hatására ezek a kapcsolatok nyitva maradnak és be tudják egymást engedni.

Mielőtt bemutatnám a STUN szerver feladatát, még meg kell említenem a NAT típusok közötti kapcsolat lehetőségeit.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Receiver | Open | Full cone | Restricted cone | Port restricted cone | Symmetric |
| Sender |
| Open | **I.** | **II.** | |  | **III.** |
| Full cone |
| Restricted cone |
| Port restricted cone |  |
| Symmetric | **IV.** | |

1. Mivel a fogadónak nincs NAT-ja, ezért a kapcsolat létre tud jönni. Hiába indul más portról minden kérése a küldőnek, a fogadóval ugyanúgy fel tudja venni a kapcsolatot mint egy publikus szerverrel, hiszen az ugyanolyan helyzetben is van mint egy publikus szerver.
2. A felek egymásra találhatnak ebben az esetben is, kivéve, ha a fogadó fél port restricted és a hívó fél szimetrikus NAT mögött van. Ennél a helyzetnél a fogadó egy invitation packet-et küld a hívónak, hogy a saját portja is nyitva legyen a hívó felé.
3. A küldő küld egy üzenetet a fogadónak. Ezzel megadva a lehetőséget, hogy a fogadó is tudjon majd neki küldeni. Ezután a fogadó egy másik csatornán megkapja a küldő fél címét és elindít egy üzenetet a küldő felé, amit ő megkap, mivel előzőleg megnyitotta a kapcsolatot a fogadó felé.
4. Általában nem megoldható a kapcsolat.

### Session Traversal Utilities for NAT (STUN)

<https://webrtchacks.com/symmetric-nat/>  
<https://tools.ietf.org/id/draft-takeda-symmetric-nat-traversal-00.txt>  
https://en.wikipedia.org/wiki/STUN

Most, hogy megismertük a NAT működését, bemutathatóvá válik a STUN, mint segédeszköz.

Röviden, egy STUN szerver segítségével a NAT mögötti kliens elég információt gyűjthet a rá érvényes NAT beállításokról, így ezt továbbíthatja magáról és kapcsolatot létesíthet más NAT mögött lévő klienssel.

A STUN szerver a TURN szerver egy egyszerű változata. Egyetlen feladata, hogy a kliensnek visszaadja, hogy a NAT-on kívül ő milyen címen és porton érhető el. Ehhez az egyszerű szolgáltatáshoz UDP-t használ, bár a TCP is lehetséges, csak ez megterhelőbb a szerverre nézve több ezer egyidejű lekérés esetén. Ezen kívül támogatott a TCP/TLS is, illetve a DNS szerver is tud szolgálni STUN címmel a megfelelő ’stun’ és ’stuns’ (TLS) rekord lekérése esetén.

Extra funkcionalitás a STUN-ban az authentikáció lehetősége.

Amennyiben a STUN szerver visszaszolgáltatta a kliens publikus ip címét és port számát, onnantól fogva az ICE protokolon múlik, hogy a két kliens egymásra talál-e.

NAT discovery: https://www.netmanias.com/en/post/techdocs/6067/nat-network-protocol/nat-behavior-discovery-using-stun-rfc-5780

### Traversal Using Relays around NAT (TURN)

A TURN arra az esetre készült, amikor nem lehet ügyeskedni úgy, mint a STUN-nal. Míg STUN esetén a kliensek egymásra találhatnak a publikus címeik alapján, addig szimmetrikus NAT esetén még ez sem lehetséges, az egyetlen kivitelezhető interakció ebben az esetben az, amikor a kliens egy publikus szerver felé kommunikál. Erre szolgál a TURN. Erőforrás igényes, mivel továbbítóként szolgál a két kliens között, azaz minden adat átfolyik rajta. Viszont meg tud bírkózni a szimmetrikus NAT-tal. (táblázat, IV. Rész)

Legelterjedtebb implementációja a coTurn, egy open-source STUN/TURN szerver implementáció. Minden szolgáltatás ezt használja, két okból is:

* C-ben íródott, gyors, alacsony fogyasztású
* A szabványokat erőteljesen betartja

### Interactive Connectivity Establishment (ICE)

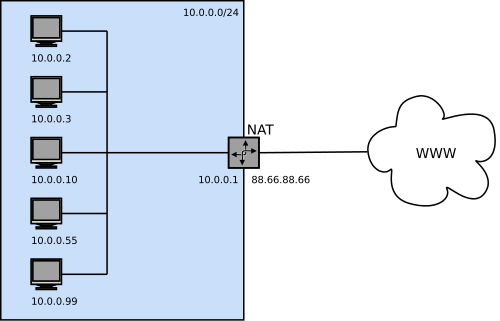
<https://www.slideshare.net/saghul/ice-4414037>

https://www.slideshare.net/rootkiskacsa/stun-turn-pocpilot?next\_slideshow=1

Mielőtt bemutatom az SDP fejlécet, amit a WebRTC kliensek generálnak, meg kell említenem pár fontos protokollt, ami képessé teszi a kommunikációt a lokális hálózaton kívül is.

A WebRTC lényege, hogy a kliensek között nyújtson összeköttetést. (Ennek előnyeiről és megoldásairól fogok írni a WebRTC fejezetben.) Viszont ez a való életben nem olyan egyszerű. Amennyiben lokális hálózaton próbálkoznánk a WebRTC fejlesztésével, nem jutna eszünkbe, hogy rendkívül kényelmes az IP címeket készpénznek venni, mint végpontok amik egy hop alatt eléretőek, de az esetek nagyrészében nem így néz ki az összeköttetés.

Vegyünk egy egyszerű példát. Alice T-Home-ot használ, Bob pedig UPC-t. Alice is NAT mögé van rejtve, Bob is. Nincs egyedi IP cím, ami alapján Alice eljutna Bob-hoz, és fordítva.



Amennyiben nem törődünk ezzel a problémával, legjobb esetben is csak egyirányú kapcsolatot kapnánk, hiszen csak aki nyitja a kapcsolatot, annak lenne nyitva a portja a válasz megkapásához.

Proxy használatával meg lehetne oldani a problémát, viszont a proxy szerverek általában fix portokon működnek, a WebRTC pedig random értékű kiosztást használ.

A problémát meg lehetne oldani kliens oldalon, de szimmetrikus NAT esetén még így se működne egymás megtalálása. Emiatt muszáj egy publikus hálózaton lévő szerver, ami segít a két kliensnek, őket hívják STUN/TURN szervereknek.

Ezek mind félig működő módszerek, azonban az ICE teljes egészében megoldja a NAT állította problémát. Tulajdonságai:

* STUN és TURN szerverek felhasználása
* TURN szerver a worst-case, hiszen ez a feladás kategória
* Csak akkor kezd adatot küldeni az adott session, amikor már teljesen felépült a kapcsolat és biztos nem lesz benne szakadás a NAT miatt
* Rendkívül komplex protokoll, 6 évbe telt, mire RFC fáziba jutott

Az ICE protokoll:

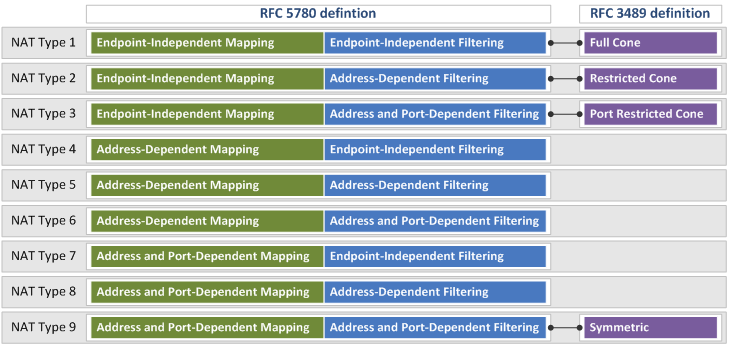
* Allokáció
  + Gyűjtsük össze az összes candidate címet. Ez lehet a gép hálózati interfészei és annak adatai, vagy a STUN/TURN szervertől tanult cím
* Priorizálás
  + Adott candidate milyen prioritást kap a lehetőségek közül
* SDP attribútum: a=candidate:2222700650 1 udp 2113937151 192.168.1.105 53352 typ host generation 0 network-cost 999
  + **2222700650:** unique id
  + **1:** RTP: 1, RTCP: 2
  + **Átvitel módja:** UDP
  + **Prioritás:** 
    - (2122260223>>24)=126 -> host candidate
    - 256-(2122260223&255)=1 -> rtp
  + **IP cím és port**
  + **Típus**
* Miután az offer (SDP header) létre jött, INVITE jelzésű üzenetként küldi el a kliens. A kliens akinek továbbítva lett a header, összegyűjti a saját candidate-jeit, rangsorolja őket és választ generál, ezt is meg kell kapnia a hívás indítójának
* Ellenőrzés
  + Mindkét fél birtokában van az SDP header.
  + Kapcsolat ellenőrőző üzeneteket küldenek egymásnak, ami abból áll, hogy a STUN szervert kérik, hogy nyisson portot nekik a másik gép felé, majd ezt ellenőrzik.

Az ICE portokoll csak akkor alkalmas oda-vissza kapcsolatra, ha maximum az egyik fél van szimmetrikus NAT mögött.

<https://www.slideshare.net/rootkiskacsa/webrtc-hol-tartunk-ma>

https://www.netmanias.com/en/?m=view&id=techdocs&no=6065

Szabványos NAT-Tűzfal átjárás biztosítása



### Session Description Protocol (SDP)

https://github.com/jbebe/webrtc-thesis/wiki/AllThingsUnsorted

https://webrtchacks.com/sdp-anatomy/

https://www.slideshare.net/saghul/ice-4414037

https://andrewjprokop.wordpress.com/2013/09/30/understanding-session-description-protocol-sdp/

https://en.wikipedia.org/wiki/Session\_Description\_Protocol

https://www.voip-info.org/sdp/

https://andrewjprokop.wordpress.com/2014/07/16/an-introduction-to-webrtc-and-signaling/

Az SDP (session description protocol) célja, hogy multimédia kommunikáció esetén információt nyújtson a másik fél számára a küldő lehetőségeiről, elérhetőségéről. Maga a protokoll nem egy folyamatos oda-vissza üzengetést ír le, sokkal inkább hasonlítható egy egyszeri HTTP fejléc request-response adatcseréjéhez, ahol megadjuk a saját meta-adatainkat illetve a küldött és várt adat típusát. Egy fél, aki előállítja saját SDP header-jét, felsorolja, hogy milyen módon lehet hozzá kapcsoldni és milyen módon tud fogadni, vagy stream-elni média tartalmat.

A fejléc nagyon egyszerűen néz ki, ebben is hasonlít egy kicsit a HTTP-hez. Teljesen plaintext alapú, a kulcsok karakterek, az értékek pedig új sort nem tartalmazó tetszőleges hosszú stringek.

Az SDP 3 fő részből áll; session, időzítés, media description. Session definícióból csak egy lehet, de időzítésből és media description-ből tetszőlegesen sok.

**SDP fejléc**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Az SDP fejléc első és legfontosabb része a session data. Ennek a következő kulcsai vannak:   |  |  | | --- | --- | | v=0 | Protokoll verziója. Jelenleg csak a 0 érték elfogadott. | | o=- 6037493519144602100 2 IN IP4 127.0.0.1 | Az első érték maga a session id. A 2-es szám a session verziójára utal, tehát ha később változtatunk a stream-en, például kikapcsoljuk a mikrofont, az újragenerált fejlécben 3-as érték fog szerepelni. Az IN az internetet jelenti, mint szállító eszköz, az IP4 természetesen az IPv4 protokollra, az utolsó ip cím pedig a gép ip címe. (Ez az utolsó három adat nem fog kelleni a WebRTC kapcsolat felépítéséhez, mivel ahhoz az ICE protokollt használjuk.) | | s=- | Az s kulcshoz a session szövegesen megadott neve tartozik, amit láthatóan nem használnak ki az implementációk. | | t=0 0 | A t a time-ra utal, ez a kulcs adja meg, hogy milyen időintervallumon valid a session. Láthatóan 0 időpillanattól 0-ig, ami jelenthetné azt, hogy soha nem érvényes, de jelen esetben ez azt jelenti, hogy tetszőleges időre szól a session. | | a=group:BUNDLE audio video data | Az a kulcs attribútumot jelent, ezen belül is a bundle group több stream együttes átvitelét jelzi, melyeket külön-külön később részletez a saját leírója. Mint látható audio, video és data stream-ek átvitelére való képességet jelez ami pont elegendő egy valós idejű chat-eléshez. | | a=msid-semantic: WMS lXRsYHOZTANYJUkp60Qfmeyc4eKkAwsaljaQ | Ez az attribútum egy egyedi azonosítót rendel a WMS-hez (WebRTC Media Stream). A stream-ek ezzel lesznek azonosítva. | |
| A következő (opcionális) rész a különböző médiatípusokat írja le:   |  |  | | --- | --- | | m=application 53350 DTLS/SCTP 5000 | Ez az attribútum írja le, hogy datastream is tartozik a session-höz, mégpedig az 53350-es porton DTLS titkosítással és SCTP protokollon keresztül. | | c=IN.... a=candidate... a=ice-... | Ezek az attribútumok hasonlóak az audio és video leírásokhoz. | |
| A következő (opcionális) rész az audio csatornát írja le:   |  |  | | --- | --- | | m=audio 58779 UDP/TLS/RTP/SAVPF 111 103 104 9 0 8 106 105 13 126 | Ez az attribútum írja le, hogy audio csatornát tud nyitni a session az adott porton. UDP-n keresztül, TLS titkosítással. Az RTP/SAVPF pedig az SRTP, SRTCP és RTCP protokollok használatát írja elő. | | c=IN IP4 217.130.243.155 | Ez a sor adja meg, hogy honnan várod az audio kapcsolatot. Mivel a pontos címet az ICE protokoll t | | a=rtcp:51472 IN IP4 217.130.243.155 | Ez a sor írja le az rtcp kapcsolatot. (A port megegyezik az SRTP porttal, mivel ezek multiplexálhatóak.) | | a=candidate:1467250027 1 udp 2122260223 192.168.0.196 46243 typ host generation 0  a=candidate:1467250027 2 udp 2122260222 192.168.0.196 56280 typ host generation 0 | Az ICE jelölt adatai. Rengeteg ilyen attribútum pár lehet az SDP fejlécben, mindegyik egy lehetséges csatornát ír le, melyen keresztül felépülhet a kapcsolat Az első komponens ahol 1-es szám van, az RTP protokollra vonatkozik, a 2-es szám pedig az RTCP-re. | | a=ice-ufrag:Oyef7uvBlwafI3hT  a=ice-pwd:T0teqPLNQQOf+5W+ls+P2p16 | E két attribútum a kapcsolat titkosságát hivatott erősíteni, tehát csak az adott jelszó birtokában kapcsolódhat a másik fél a hoszthoz. | | a=fingerprint:sha-256 49:66:12:17:0D:1C:91:AE:57:4C:C6:36:DD:D5:97:D2:7D:62:C9:9A:7F:B9:A3:F4:70:03:E7:43:91:73:23:5E  a=setup:actpass | Ezek az attribútumok a DTLS kapcsolathoz kellenek, a hash pedig a cert alapján készül. | | a=mid:audio | Ez az attribútum a bundle-ben megnevezett csatornára utal, tehát a média leírását köti a globális konfigban lévő hivatkozáshoz. | | a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level  a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time | Ezek az attribútumok az RTP header-t egészítik ki a megadott módon metaadatokkal, tehát a másik félnek is támogatnia kell ezeket a kiegészítéseket. | | a=sendrecv | Fontos attribútum: ez jelzi, hogy a header kiállítója az adott csatornát küldi, fogadja, mindkettő, vagy egyik sem. Például egy előadás esetén a stream-et sendonly és recvonly módban érdemes használni. | | a=rtcp-mux | Ez a sor írja elő, hogy az RTCP és RTP kapcsolat egy porton keresztül fog menni. | | a=rtpmap:111 opus/48000/2 | Ez a sor az opus codec-et írja elő az átvitelhez. Mivel ingyenes és szabadon felhasználható, ezért ez az egyik legelterjedtebb codec. | |
| A video kapcsolat hasonló módon van előírva, csak ott természetesen más codec-eket használ a WebRTC. |

### WebRTC technológia

<https://www.slideshare.net/rootkiskacsa/webrtc-hol-tartunk-ma>

Különböző definíciói vannak a WebRTC-nek.

* Protokoll nagy felbontású, sok képességgel rendelkező RTC alapú programok fejlesztéséhez minden platformra.
* Framework, protokollok és API-k összessége ami lehetőséget nyújt audio video és adatátvitelhez.
* Különböző platformok számára nyújt RTC képességet. A WebRTC ennek optimalizált megoldása.

Előnyei:

* Nem kellenek hozzá plugin-ek
* Nem kell hozzá telepítés
* Peer 2 peer működés
* Tűzfalon való átjárás
* WebAPI elrejti a komplexitást (A WebAPI-t pedig a lib-ek rejtik el)
* Authentikáció, Biztonság
* Implementáció elérhető nem csak a böngészőben, de mobilokra is, natívan is.
* Tetszőleges signaling protokoll
* Alacsony késleltetés (UDP)
* Torlódásvezérlés

Érdekességek:

* 2016-2017: 45% növekedés használatban
* 1300-nál több komoly projekt
* Böngészők 80%-a WebRTC ready

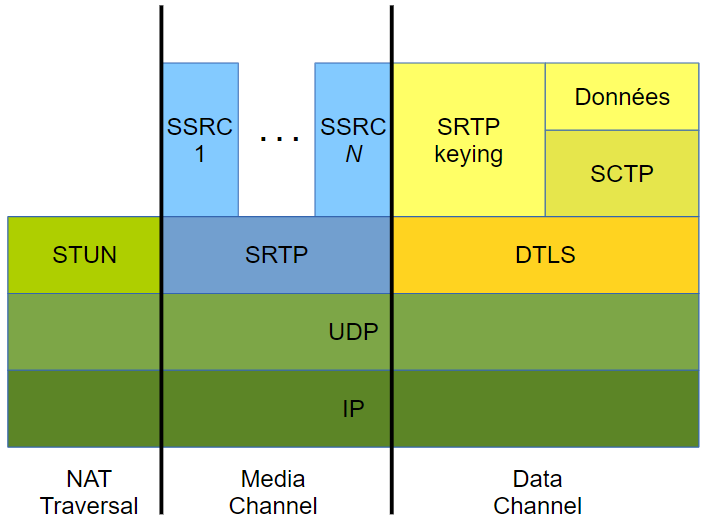


Figure 5. NAT-hoz tartozó protokoll, kép és hang küldéséhez használt stack, adat küldéshez használt stack



Figure 6. Mindez részletesebben

Idővonal:

* Google-nél megszületik az ölet
* 2010 nyarán Google Microsoft Apple Mozilla Skype Ericcson összeül
* 2011-ben már Chrome támogatás
* 2013 – Firefox támogatás
* 2015 – Edge
* 2017 – feature freeze, 1.0 szabvány

Kodekek:

* Opus és G711
* VP8 vs. H.264
* AfOM

Titkosítás

* SRTP-DTLS
* Személyazonosság

Gateway

* RTPEngine
* Janus
* Kurento
* FreeSwitch
* JITsi gateway to SIP

https://webrtchacks.com/trickle-ice/

## WebRTC a gyakorlatban

Ebben a fejezetben a WebRTC gyakorlati hasznosításáról fogok beszélni. Először a használható csatornákról, majd a kliensek közti topológiáról, végül a média szerverek használatáról, azok különbségeiről.

<https://www.slideshare.net/rootkiskacsa/webrtc-puzzle>

Adapter.js

* Implementációs hézagok áthidalása
* Szabványosítás, implementáció

### Lehetőségek

#### Média

A böngészőben többféle média tartalmat tud a WebRTC küldeni és fogadni. Tulajdonképpen bármely stream típusú erőforrás megfelelő neki, így a webkamera és mikrofon alapból támogatott. Most bemutatom, hogy konkrétan milyen tartalmakat lehet átvinni WebRTC kapcsolattal.

**Webkamera:** A különböző médiatartalmat stream-elő eszközöket a

navigator.mediaDevices.enumerateDevices()

hívással tudjuk lekérni. Itt mindenféle input és output eszközöket felsorol a böngésző amik elérhetőek számára. Ezek vagy audio vagy video típusúak lehetnek. Ha például csak a webkamera video részét szeretnénk elkérni, azt a következő módon tehetjük meg:

navigator.mediaDevices.getUserMedia({ video: true })

Egy adott médiához tartozhat egy úgynevezett constraint objektum, ami leírja, hogy az adott eszköznek milyen konfigurációjában szeretnénk felhasználni a stream-jét. Audio esetén ez lehet a mintavételezési frekvencia, hangerő, visszhang kioltás, stb. A teljes listát lekérhetjük az eszközötől, így tisztában lehetünk a támogatott szolgáltatásokkal mielőtt azokat elkezdenénk használni.

<https://developer.mozilla.org/en-US/docs/Web/API/MediaTrackConstraints>

**Képernyő megosztás:** Bizonyos böngészőkben (Chrome, Firefox) lehetőség van a teljes képernyő, egy program, vagy csak egy honlap képének megosztására. Tegyük fel, hogy stream-elni szeretnénk az ismerőseinknek ahogy játszunk. Ekkor a következő függvényt kell meghívnunk:

navigator.getDisplayMedia({ audio: true, video: true })

**Canvas:** Az ezt támogató böngészőkben (Chrome, Firefox) lehet <canvas> objektumot stream-elni, tehát lehetőség van rajzolni a másik félnek valós időben, vagy akár egy WebGL-lel futó játékot továbbítani.

const stream = canvas.captureStream();

lásd: <https://webrtc.github.io/samples/src/content/capture/canvas-pc/>  
[https://docs.google.com/document/d/1JmWfOtUP6ZqsYJ--U8y0OtHkBt-VyjX4N-JqIjb1t78](https://docs.google.com/document/d/1JmWfOtUP6ZqsYJ--U8y0OtHkBt-VyjX4N-JqIjb1t78/)

**Egyéb:** Lehetőség van a stream-ek felvételére is, bár ezt nem igényli a WebRTC, mint inkább egy chat szolgáltatás esetén a felhasználó. A következő módon vehetjük fel a stream-et és menthetjük el:

const mediaRecorder = new MediaRecorder(window.stream, options);

mediaRecorder.ondataavailable = (event) => recordedBlobs.push(event.data);

mediaRecorder.start(1000/\*ms\*/);

const videoBlob = new Blob(recordedBlobs, {type: 'video/webm'});

#### Adatfolyam

A WebRTC tetszőlegesen sok adatfolyamot enged meg a média stream-elésen kívül. Ez az adatfolyam kezdeti beállítástól függően lehet veszteséges, illetve veszteségmentes és sorrendezett.

A folyamot az indító kliens hozza létre a createDataChannel függvénnyel. A fogadó csak a kapcsolatának ondatachannel eseményénél értesül erről, így a fogadó a kapcsolat elején még nem tudja hány csatorna fog felépülni köztük.

A folyamon a send() metódussal lehet küldeni adatot. Stringet, Blob-ot (ami a fájlnak felel meg böngészőben) illetve ArrayBuffer-t.

Fájl küldés esetén lehetőség van annak bufferelt küldésére, az ehhez szükséges metaadatokat lekérdezhetjük a böngészőben, mint peldául az eddig elküldött byte-ok számát, vagy a buffer méretét.

### Topológia

A WebRTC kapcsolat egy-egy összeköttetést jelent, azaz csak egy hívó és egy válaszoló fél között jön létre. Ahhoz, hogy több emberrel, például egy konferencia beszélgetést folytassunk, lehetőségünk van különböző opciók közül válogatni. Mindeközben figyelemmel kell lennünk az elfoglalt sávszélességre, illetve a böngésző által kezelhető stream-ek számára.

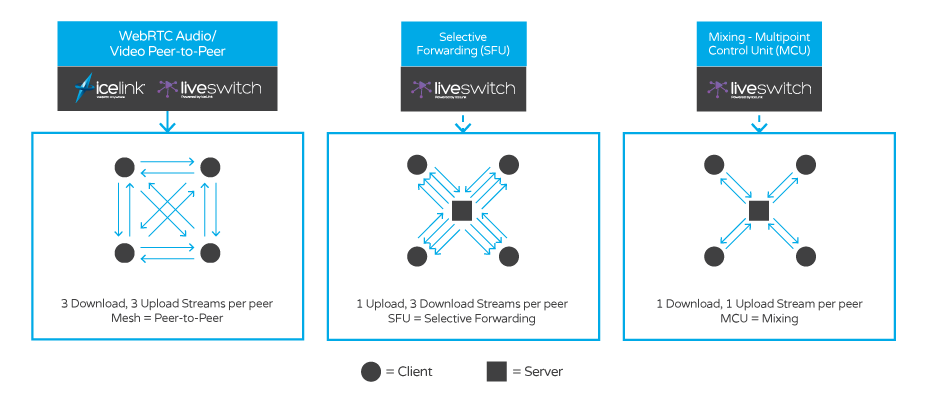


Figure 7

#### Egy-egy

Ez a legegyszerűbb eset. A hívó fél kapcsolódik a hívott félhez. Párhuzamos videoközvetítés esetén ez 1 video stream feltöltését és 1 letöltését jelenti. Az architektúra szempontjából ez a legideálisabb állapot. Az átvitt adat száma itt 2.

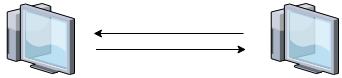


Figure 8

#### Egy-több

Ha valaki egy előadást szeretne sugározni, ez a topológia kell neki. Mondhatnánk azt, hogy ez az előző példával ellentétben csak 1 adat átvitele, mivel a nézők nem küldenek vissza adatot. Sajnos az ISP-k nem támogatják a multicast-et, és azzal sem lennénk előrébb, ha minden fogadó ugyanazt az SDP offert fogadná el. Emiatt az N vevő miatt N adat átküldése szükséges.

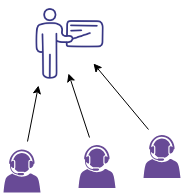


Figure 9

#### Teljes gráf

Mivel a WebRTC egyéb segítség nélkül pont-pont kapcsolatot tud csak kiépíteni az SDP segítségével, ezért egy több szereplős kommunikáció esetén teljes gráfot kell kiépítenie a kliensek között. Ez 2 résztvevő esetén még nem túl nagy overhead, mivel csak 1 stream-et küldünk és fogadunk, de ennél több kliens kiszolgálása már komoly megterhelést jelent a hálózat számára.

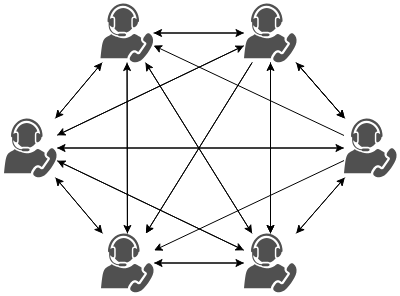


Figure 10.

A felső ábra jól mutatja, hogy a stream-ek számának növekedése polinomiális, egész pontosan egy teljes gráf éleinek duplája. Mivel minden résztvevő a csoport összes többi tagjának küld üzenetet, ezért az odafelé vezető kapcsolat nem helyettesíthető a visszafelé vezető kapcsolattal, így kapjuk az N\*(N-1) összefüggést a stream-ek száma és a részvevők száma között.

Előnye, hogy a módszer nem igényel szervert, így teljes egészében a klienst terheli a kapcsolat. Ezen kívül szerver hiányában a leggyorsabb reakcióidő érhető el vele. Amennyiben fontosabb nekünk a reakcióidő és az olcsó szerver költségek, ez az egyetlen módszer amit választhatunk.

Hátránya, hogy nem mindegyik kliens képes feldolgozni ugyanannyi különálló stream-et, illetve a sávszélesség is kritikus pontja a sok stream átvitelének.

### Selective Forwarding Unit (SFU)

* JITSI video bridge
* Kurento Media Server
* Janus
* MediaSoup
* Licode Erizo

<https://bloggeek.me/webrtc-sfu-challenges/>

<https://bloggeek.me/media-server-for-webrtc-broadcast/>

<https://www.slideshare.net/mganeko/webrtc-sfu-mediasoup-sample>

Amennyiben olyan kommunikációs szolgáltatást szeretnénk üzemeltetni, ahol nem limitálhatjuk a résztvevők számát háromra, szükségünk lesz egy plusz szolgáltatásra, ami megoldja azt, hogy a stream-ek ne haladjanak redundánsan a kliensek között. (emlékezzünk a teljes gráf alakú N résztvevős kapcsolatra) Erre jelenthet megoldást az SFU, vagy az MCU. Most az SFU-t mutatom be, ami a szerver oldaláról nézve kisebb számítást igényel.

Az SFU egy szerver, ami köztes félként szolgál a kliensek között. Összegyűjti a kliensektől érkező stream-et és a többi résztvevőnek továbbítja. Így a teljes összeköttetés esetén igényelt 2(N-1) stream helyett már csak N stream-et kell kezelnie egy kliensnek, hiszen N-1 érkezik és 1-et továbbít felfelé.

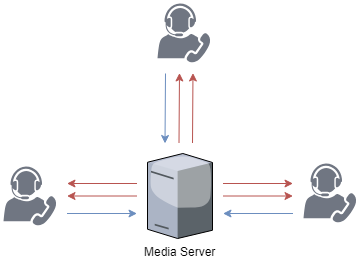


Figure 11. Az SFU topológiája

Egy újabb megközelítés született az SFU területén, hogy a renderelést elkerülhesse a szerver, de a stream-ek különböző méretben juthassanak el a kliensekhez. Ennek a neve Simulcast. A kliensek több felbontásban küldik el a stream-et az SFU-nak, amaz pedig azt a változatot küldi tovább a többi résztvevőnek, amit azok éppen preferálnak. Ha például A kliens az előadó és B C D a nézők, akkor elég B C és D-nek csak a kis felbontású képét küldeni a többi résztvevőnek. Vagy például az alapján is lehet válogatni a felbontások között, hogy kinek éppen mekkora sávszélességű internete van.

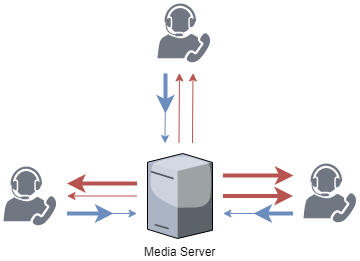


Figure 12. SFU optimizálva

Az SFU implementációja általában úgy néz ki, hogy a szerver oldal mellett egy kliens oldali könyvtárat is készít a fejlesztő csapat, ami kezeli a kapcsolatot a szerver felé, mi pedig már csak kész stream-eket kapunk a kapcsolódási eseményre.

### Multipoint Control Unit (MCU)

* FreeSwitch
* Kurento Media Server
* Medooze MCU Media Server
* Licode Erizo

<https://www.slideshare.net/Dialogic/beyond-the-mcu>  
https://images.tmcnet.com/expo/webrtc-conference/presentations/san-jose-14/D3-4-Media-Services.pdf

Az SFU-val szemben az MCU sokkal megterhelőbb számítási igénnyel rendelkező megoldás. Ilyenkor a kliens az SFU-hoz hasonlóan egy könyvtár segítségével kapcsolódik a szerverhez és szintén azon keresztül értesül az új résztvevők érkezéséről. Előbbivel ellentétben viszont nem az összes többi kliens stream-jét küldi vissza az aktuális kliensnek, hanem egy összerenderelt, kompozit stream-et.

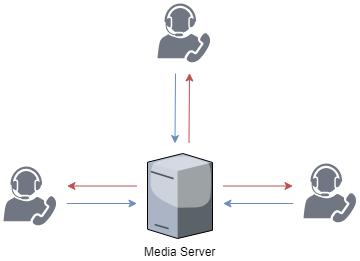


Figure 13. MCU szerver

Előnye, hogy ilyenkor a kliensek már tényleg csak egy felfelé és egy lemenő stream-et kezelnek, ami nagyon előnyös, hiszen kisebb teljesítményű eszközökön, rosszabb átviteli képességgel rendelkező hálózaton is jobb minőséget biztosít a többi megoldással szemben.

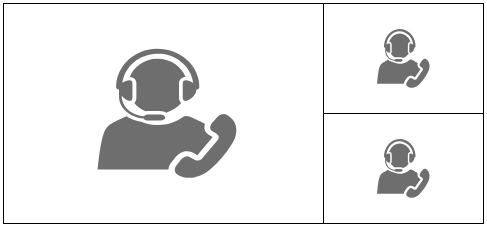


Figure 14. Előre renderelt képernyő elrendezés

Előnye ezen kívül még az is, hogy a szerver – mivel a teljes visszaadott stream-et újrakódolja – bármilyen extra munkát elvégezhet a stream-en, anélkül, hogy a kliensnek ez munkájába kerülne, például effektek, vagy a kliensétől különböző kodek használata, hang és kép élesítése.

Előny és egyben hátrány is, hogy az elrendezése a visszajövő stream-nek kötött, így nem mozgathatjuk őket tetszőlegesen, úgy mint SFU esetén, hanem kompromisszumot kell kötnünk, a UI design része kell, hogy legyen az egymás mellett elhelyezett ablakok.

Gondoljunk bele, hogy egy chat szolgáltatás esetén a többezer párhuzamosan kommunikáló felek mekkora forgalmat generálnak. Ha mindezt a sok stream-et egy szervernek még össze kell renderelnie 3-4-5-ösével, akkor annak hatalmas teljesítménybeli hatása lenne. Így érkeztünk meg az MCU legnagyobb hátrányához, mégpedig a stream-ek rendereléséhez. Az SFU-nak nem szükséges hozzányúlnia a stream-ekhez, az MCU-nak viszont muszáj, hiszen egy képet akar renderelni belőle. Ez pedig nagyon sok CPU műveletet jelent, amit a szolgáltatás üzemeltetőjének kell fizetni.

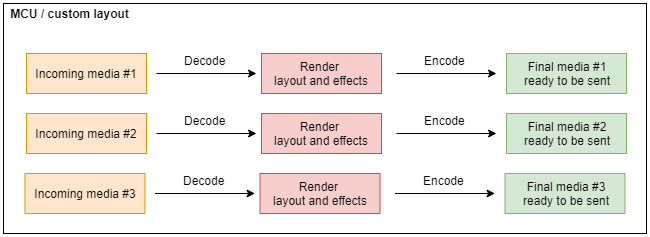


Figure 15. Legnagyobb terhelés az MCU szerveren

Egy hatékonyabb módszere a az MCU felhasználásának az, ha kihasználjuk a stream valósidejűségét és ahelyett hogy a kliensek határoznák meg a stream elrendezését, csak egyet renderel az MCU és azt küldi le minden félnek. Ez sokkal hatékonyabb, ám nyilvánvalóan a kliens szabadságának rovására történik.

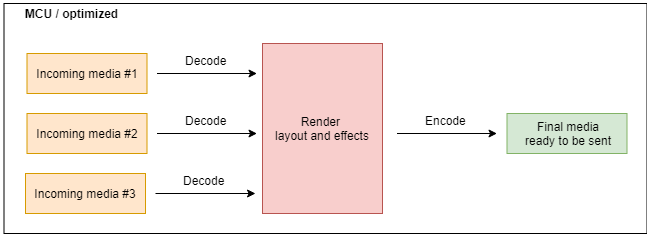


Figure 16. Optimalizált renderelés az MCU szerveren

Ennek a közös renderelésnek az eredménye, hogy sokkal kevesebbet kell számolnia a processzornak. Az alábbi grafikon mutatja, hogy mennyivel effektívebb az elrendezés fixálása és közös rendelése:

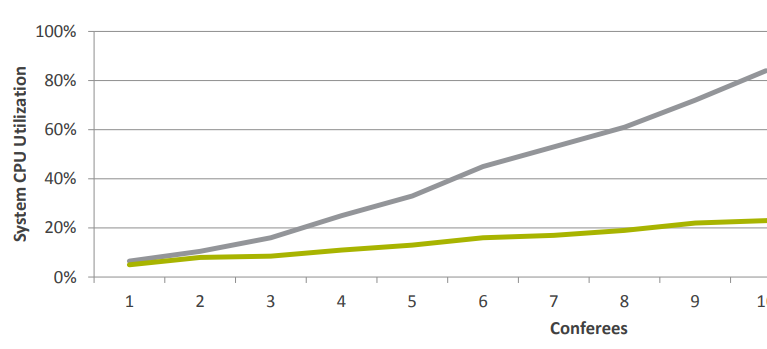


Figure 17. A zöld vonal mutatja a közös renderelést.

### Mesh vs. SFU vs. MCU

Most, hogy bemutattam az SFU és MCU hátrányait és előnyeit, vessük össze, hogy egymáshoz képest milyen hozományaik és hátrányaik vannak.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Teljes gráf | SFU | MCU |
| Renderelés okozta késleltetés | - | - | Számottevő, egy layout esetén kevésbé |
| Hálózati terheltség (szerver) | - | Közepes  (N-1 stream / client) | Alacsony (1 stream / client) |
| Hálózati terheltség (kliens) | Magas  (2(N-1) stream ) | Közepes (N stream) | Alacsony (2 stream) |
| Videó elrendezése | Szabadon rendezhető | Szabadon rendezhető | Fix kereten belül / optimalizált esetben teljesen fix elrendezés |
| Késleltetés | Alacsony | Közepes | Nagy |
| Kompatibilitás | Nagy | Közepes | Nagy |

https://bloggeek.me/webrtc-multiparty-video-alternatives/

## Létező megoldások

#### SFU

<https://github.com/versatica/mediasoup>

A mediasoup egy megvalósítása az SFU-nak. Ilyenkor elfelejthetjük a saját WebRTC implementációnkat, mert egy komplett szobát kapunk, aminek eseményeire feliratkozva magasabb szinten, stream-ekként kezelhetjük a csatlakozó klienseket. Hátránya, hogy nincs lehetőségünk a finomhangolásra, előnye pedig ugyanez, mivel kisebb az esély a hibára, így, hogy egy kész implementációt kapunk.

A követkető kódrészlet a mediasoup SFU használatát mutatja be:

const room = new mediasoupClient.Room();  
room.join(username)

.then(handlePeersAlreadyInRoom)

.then(addOwnStreamToRoom);

channel.on('message', handleNewMessage);

Mint látható, sehol nem kell a WebRTC apit használni, ez mind el van rejtve a kliens elől, csak szobákat menedzselhetünk.

#### Scalable Video Coding

<http://info.vidyo.com/rs/vidyo/images/WP-Vidyo-SVC-Video-Communications.pdf>

#### Last N

<https://jitsi.org/wp-content/uploads/2016/12/nossdav2015lastn.pdf>

A Last N módszer az SFU-n alapszik és célja, hogy nagy mennyiségű beszélgető fél esetén se igényeljen túl nagy CPU használatot illetve hálózati forgalmat. Lényege, hogy az összes résztvevőből csak egy annál kisebb, ideális méretű halmaz video stream-e látszik, a többieknek csak az audio összeköttetése állandó. Így megoldottuk azt a problémát, hogy egy konferenciabeszélgetésnél az összes video stream-et továbbítani kelljen. Az N darab slot-ba sorolás beszéd alapján történik, tehát amikor valaki éppen beszél, bekerül egy látható slot-ba, aki pedig a legrégebb óta nem szólalt meg onnan, kikerül. Úgy kell ezt elképzelni, mintha egy pódiumon az emberek váltogatnák egymást aszerint, hogy éppen ki szólal fel.

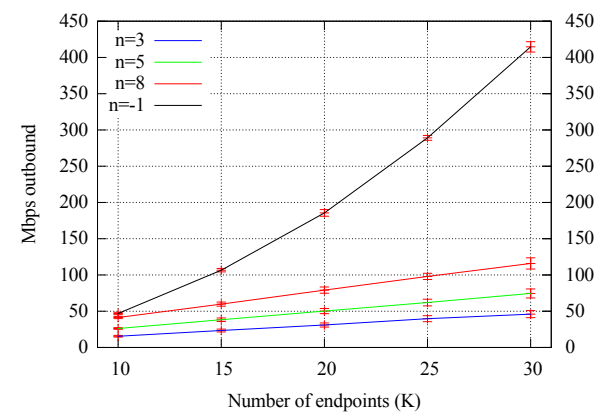


Figure 18. Sávszélesség csökkenése Last N használatával

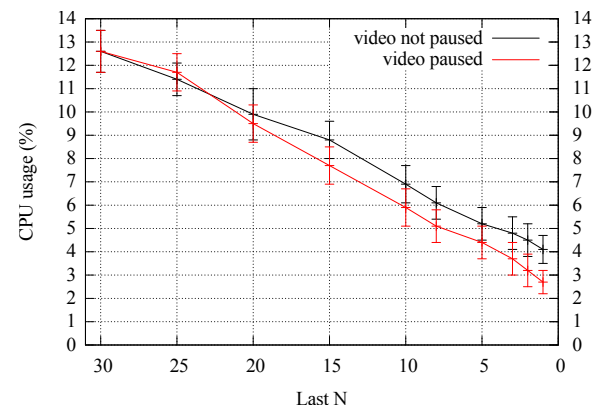


Figure 19 CPU használat csökkenése a slot-ok csökkentésével

Természetesen ehhez kliens oldalon fel kell dolgozni a résztvevő hangját és decibel alapján eldönteni, hogy beszél, vagy sem. A média szervernek ezen kívül kontroll üzenetet is kell küldeni, tájékoztatva arról, hogy a kliens nem beszél, tehát nem fog stream-et küldeni. Az emberi kommunikáció közel szimplex mivolta lehetőséget ad így a további optimalizációra.

#### Facebook

<https://webrtchacks.com/facebook-webrtc/>

https://www.youtube.com/watch?v=F7UWvflUZoc

A Facebook már egészen korán, 2015-ben elkezdte használni a WebRTC-t, mint technológiát. Architektúrájuk a következőképpen néz ki:

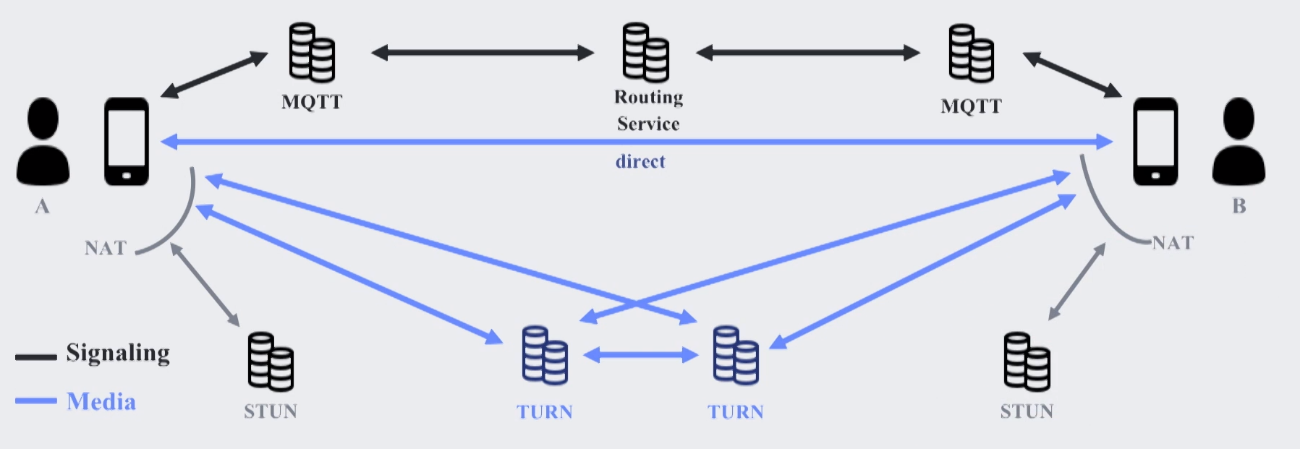


Figure 20. Facebook Messenger architektúra

Mint látható, MQTT-t használnak az SDP átvitelére, azaz a signaling-hoz. Ez a valóságban azt jelenti, hogy websocket fölött kerül átvitelre az üzenet MQTT-kompatibilis API-n keresztül. Ezen kívül használnak STUN és TURN szervereket. A következő ábrán látszik, hogy a Facebook is az SFU mellett tette le a voksát, ha több ember kommunikációjáról van szó.

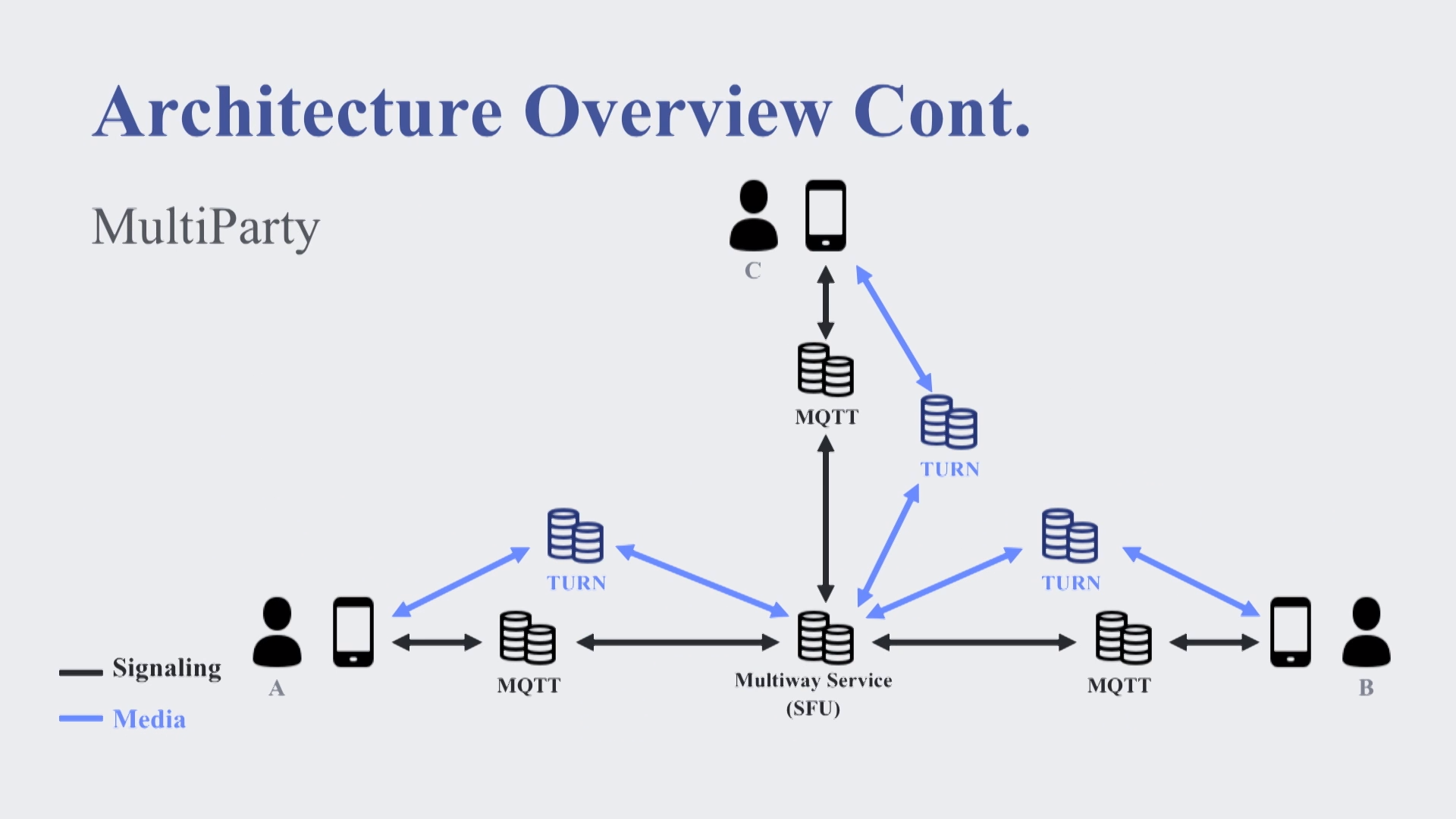


Figure 21.

#### Skype for web

https://www.youtube.com/watch?v=42hIJGFmedw

A Skype az egyik legismertebb video chat alkalmazás, mégis a böngészőből való működés csak 2015-ben vált elérhetővé. Konzervatív döntés született, mivel még ebben az évben is inkább az általuk írt plugin-re hagyatkoztak, nem pedig a gyerekcipőben járó WebRTC-re, vagy ORTC-re. Most, 2018-ban már alig pár böngésző támogatja a plugineket, így a Skype-nak is lépnie kellett, és egy ideje ők is WebRTC-t illetve ORTC-t használnak.

# Elkészült rendszer bemutatása

Most, hogy bemutattam a WebRTC technikai hátterét illetve alkalmazását konkrét termékekben, ideje prezentálnom a saját munkámat. Az eddig összeszedett tudás akalmazásával létrehoztam egy egyszerű kommunikációs csatornát, majd ennek bővítésével egy több résztvevős chat alkalmazást. Ebben a fejezetben ezeket mutatom be.

## Célkitűzések

Célom első körben a WebRTC API kipróbálása, működésének megértése, majd ennek a tudásnak a bővítésével egy több résztvevős video chat alkalmazás létrehozása volt. Szigorú, előre definiált elvárásoknak nem kellett megfelelnie, legfőbb cél a lehetőségek kiaknázása volt.

## Tervezési szempontok

Fő szempontom a proof of concept megvalósításánál a szolgáltatások legnagyobb mértékű kihasználása volt. Később a több résztvevős chat szolgáltatásnál a cél a komponens alapú felépítés, ezáltal a könnyű bővíthetőség és a stabil működés volt a fő szempont.

## Egy-egy kapcsolatú demó

### Működés

A proof of concept 3 minimalista aloldalt rejt. Egyrészt a főoldalt ahol választhatunk aközött, hogy mi akarjuk létrehozni a csatornát, vagy valaki már létrehozta, csak csatlakozni akarunk hozzá, másrészt e két funkciót betöltő aloldalt.

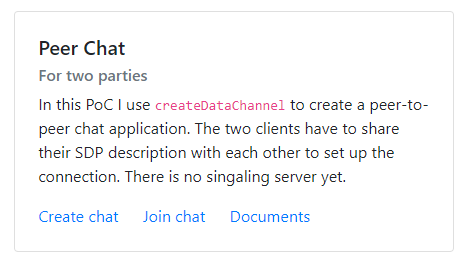


Figure 22. A kezdő képernyő

Miután kiválasztottuk a chat létrehozása menüpontot, az oldalon több lehetőségünk is van. A logger ablakban láthatjuk az összes WebRTC-vel kapcsolatos eseményt, ez az oldal felső részén helyezkedik el. Mellette kissé balrább a nyílra kattintva visszaléphetünk a főoldalra. Miközben az oldal nyitva van, a log alatt a bal panel láthatóan tölt. Ennek az az oka, hogy az SDP header még nincs készen. Ahhoz, hogy a proof of concept oldalon elég legyen egy header-t átküldeni a másik félnek, a program megvárja míg a candidate keresés - a kapcsolat felépítésének legtöbb időt igénylő része – befejeződik.

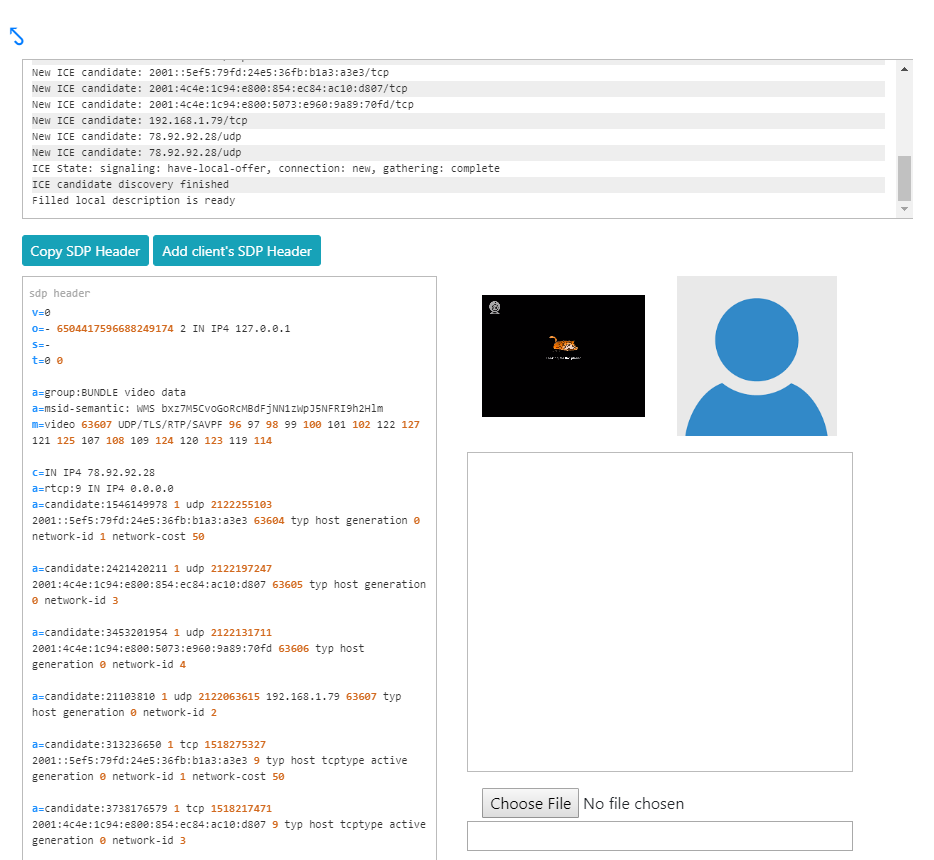


Figure 23. Host SDP létrehozása

Mellesleg a leghamarább akkor tudunk arról, hogy több candidate nem lesz, ha kapunk egy olyan eseményt OnNewICECandidate esetén, ahol a paraméter null. Amikor már az összes candiate rendelkezésre áll, az SDP header megjelenik a bal panelen. Ahhoz, hogy felhasználjuk, nem kell kijelölni, elég ha egyszerűen a “Copy SDP Header” gombra kattintunk. Ekkor a teljes tartalma a vágólapra másolódik, azzal a kiegészítéssel, hogy ez a kimásolt változat már json formátumú adat. A headert amit kaptunk most el kell küldenünk egy másik kliensnek aki szintén megnyitotta az oldalt, csak a “Join chat” gombra kattintott. Neki ugyanezen oldalon nem töltődik be megnyitás esetén a bal panelen a header, mert előbb szüksége van az offer típusú headerre, amit éppen most generáltunk. Tehát valamilyen médiumon keresztül eljuttatjuk az információt a másik félnek, aki az “Add client’s SDP Header” gombra kattintva beimportálja azt. Lokális hálózaton STUN szerver nélkül ez a header egészen kicsi, ezért nem érdemes külön fájlként kezelni, ezért döntöttem a szöveges másolás mellet. Természetesen több hálózati kártyával illetve STUN szerver segítségének igénybevételével az SDP header egészen nagy lehet amit már nem lehet könnyen kezelni, például átküldeni egy chat programon keresztül.

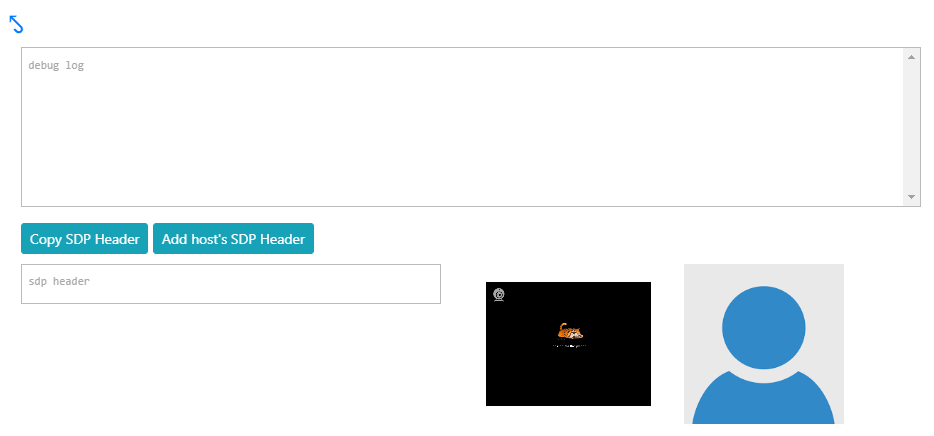


Figure 24. Guest oldal mielőtt hozzáadnánk az offert

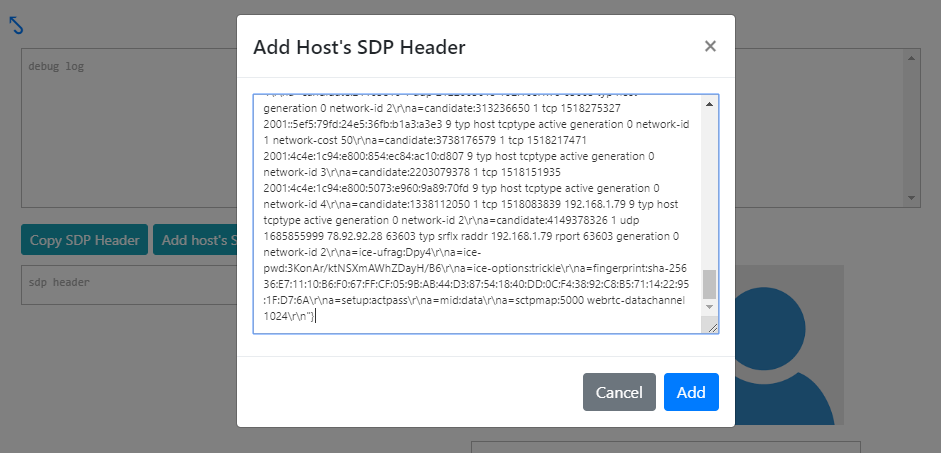


Figure 25. SDP header hozzáadása guest oldalon

Miután beimportálta a fogadó fél a headert, az ő WebRTC kliense kezd el ICE candidate-eket keresni. Ez is eltarthat egy ideig, végül megszületik az “ANSWER” típusú SDP header. Ezt szintén kimásoljuk, ahogy előbb is tettük, és átadjuk a chatet indító félnek. Ezzel elvégeztük a ránk eső feladatot. Ekkor az indító fél az answer headert az “Add client’s SDP Header” gombra kattintva véglegesíti az információcserét (signaling). Ekkor az indító fél oldaláról indulva kiépül a csatorna a két fél között. A második webkamera kép is megjelenik és a felek tudnak chatelni.



Figure 26. A létrejött kapcsolat

### WebRTC könyvtár

A WebRTC könyvtár három részből áll. Van egy base és külön a Host és Client, aszerint hogy ki hív és ki válaszol.

A Base gondoskodik arról, hogy a datachannel, videó, audio kapcsolatok létrejöhessenek, illetve kezeli az ICE candidate keresést. A Host és Client osztályok már csak a base-ből származnak le és használják fel a képességeit, hogy a saját specifikus függvényeiket ezzel együtt hajthassák végre. A Host a datachannel létrehozásáért felel, illetve az offert állítja elő, míg a Client már csak event-ként kapja meg a datachannelt és az answer-t is csak az offer megléte esetén állíthatja elő.

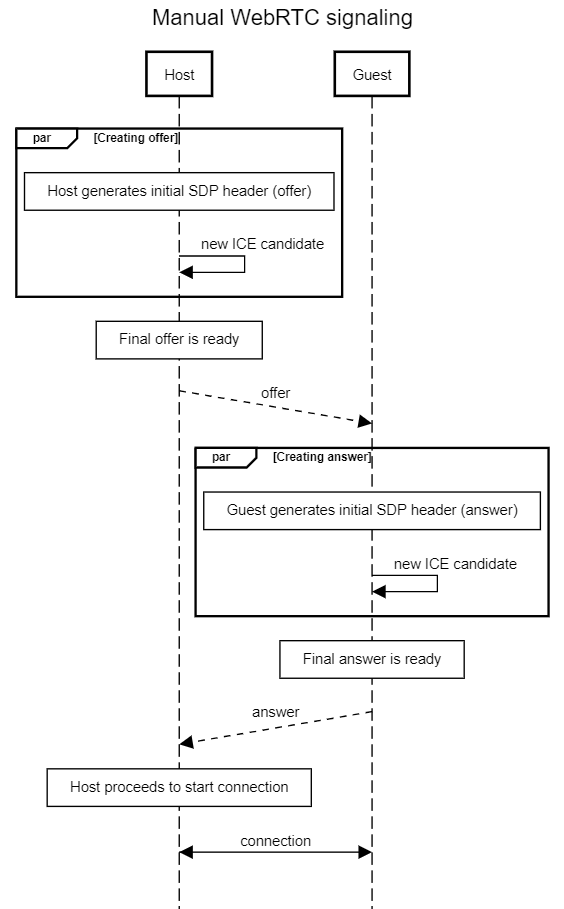


Figure 27. Kapcsolat kiépülése a PoC-ben

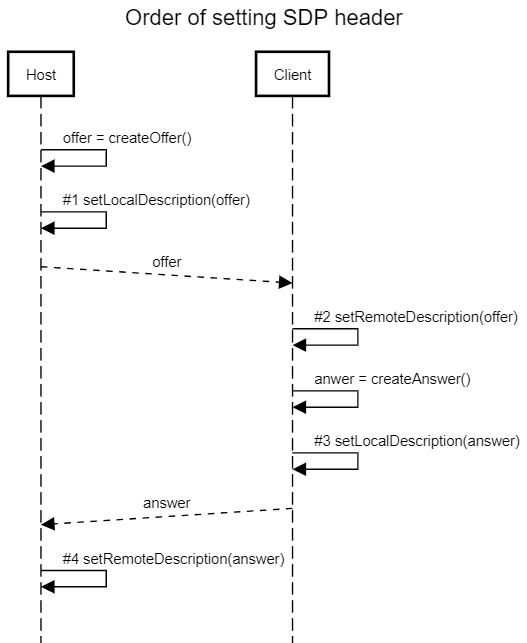


Figure 28. Az SDP header beállításának sorrendje

Ennek a könyvtárnak a megírásánál még nem figyeltem a komplexitásból eredő problémákra, hiszen egy kis könyvtárról van szó, így az ES5 nyelvi készletét használtam, teljesen mellőzve az újabb JS verziók adta lehetőségeket. Például nem megfelelően használtam a Promise-t, nagyon ritka esetben használtam await-et, helyette mindig a .then() metódust használtam ami nem segített érthetőbbé tenni a kódot. Összességében használható a könyvtár, ám a későbbi iterációk sokkal átgondoltabbak lettek. Reflektálok ezzel arra a kódrészletre, ahol egy Promise a setInterval segítségével vár arra, hogy a lokális SDP-t tároló változó értéket kapjon, ezáltal információt szerezzünk arról, hogy az ICE gathering state befejeződött. Ezen kívül nincs egy olyan async függvény, aminek visszatérésével megbizonyosodhatunk arról, hogy a kapcsolat létrejött, márpedig ez fontos elvárás egy összetettebb webapp-nál.

### Fájl küldés

Egyetlen módja a fájlok küldésének WebRTC esetén a datachannel használata. Datachannelt használhatunk bármire, hiszen a fogadott paramétertípusok a következőek: string, Blob, ArrayBuffer, ArrayBufferView. Én a PoC esetén egy ennél egyszerűbb módszert használtam, mégpedig azt, hogy egy bizonyos jelzést hagytam az üzenet elején ami alapján el lehet dönteni, hogy az éppen fájl küldés lesz, vagy csak szöveges üzenet. Természetesen használhattan volna akkor már sorosított json-t, melynek típus property-je dönti el, hogy mi az átvitt payload, de nem tettem. Későbbi tervként szerepel ennek implementálása.

### Összefoglaló

A proof of concept megírása segített megérteni a WebRTC működését. Ez alapján el tudtam indulni, hogy létrehozzak egy komplexebb rendszert. Az volt a tervem, hogy ezután már előre megírt WebRTC könyvtárat fogok használni, de később kiderült, hogy a sajátomat kellett volna folytatnom, nem nyerek máséval, sőt, nehezebb lesz kiegészíteni extra funkciókkal. Ennek oka az, hogy senkinek a piacon nem érdeke egységesíteni egy WebRTC könyvtárat, helyette csak saját specifikus implementációkat találunk, vagy egy-két olyat, ami nem sokat segít a komplexitás elfedésében.

## Több-több kapcsolatú webapp

### Bevezető

Ahhoz, hogy a megírt proof of concept működhessen a kényelmetlen SDP header manuális cseréje nélkül, először is egy szerverre van szükség. Mint azt a WebRTC bemutatásánál is említettem, a WebRTC ránk bízza a signaling-ot, ami annyit tesz, hogy az SDP cserét nekünk kell lebonyolítanunk. Ezen kívül ha már egy szerver is képbe került, session-öket is tarthatunk számon, különböző felhasználókat, azok authentikációját, stb. Magyarán egy szerverrel az előbbi demó lehetőségeit szabadon kibővíthetjük kényelmi funkciókkal, melyek egy akár eladható termék főbb feature-ei is lehetnek.

### Első verzió

Két verzió készült, amelyből az elsőt mutatom be, majd a következő fejezetben a másodikat. Megvalósításban eltérnek egymástól, de így is a hasonló részeket megpróbálom egy helyen részletezni.

#### Szerver

Az első változat egy Node.js alapú websocket szerver volt TypeScript nyelven írva. Két feladata van, egyrészt az SDP cserét jelzi a kapcsolódó felek felé, másrészt a felhasználók státuszát, elérhető felhasználók listáját küldi vissza. Ennél az elgondolásnál az egyszerűség volt a fő szempont. Nem akartam több különböző szervert futtatni, hanem egy kompakt, mindenre képes apró szerverre volt szükségem.

Az architektúra nagyon egyszerű. Egyrészt van egy statikus fájl kiszolgáló, ami az SPA-t tölti be, illetve maga a signaling server ami websocketre épül. Mind a fájl kiszolgáló, mind a WS könyvtár az express.js http szerver könyvtárat használja. Websocket-hez a socket.io könyvtárat használtam.

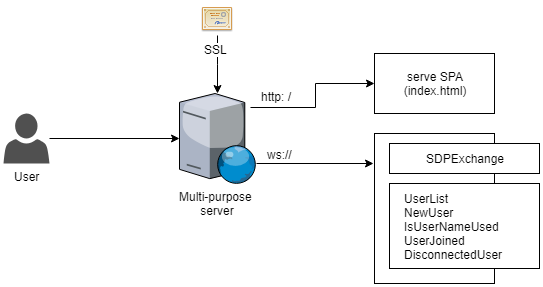


Figure 29. A kompakt szerver

A különböző WS kéréseket egy egyszerű dispatcher kezeli:

|  |
| --- |
| (router[clientMessage.type] || onError).call(router); |

Ha létezik a bejövő üzenet típusában megadott metódusnév, akkor az fut le, ha nincs ilyen, akkor pedig az onError kezeli le. Az éppen aktív felhasználókat egy User[] tagváltozó tárolja, ezért a szerver leállásával elveszik minden eddigi adat. A későbbi implementációban ez a probléma is orvosolva lett.

Jó döntésnek tekinthető a Node.js abban a tekintetben, hogy ugyanazon a nyelven íródott, mint a kliens, ezért nem olyan nehéz a context switch. Ezen kívül egy közös fájlban tárolható minden típus amit a szerver és a kliens használ ami megkönnyíti a fejlesztést és kisebb az esélye a hibázásnak. És végül azért is jó, mert a Node.js jól kezeli a gyors aszinkron műveleteket, amiből a WS szerver miatt éppen sok van.

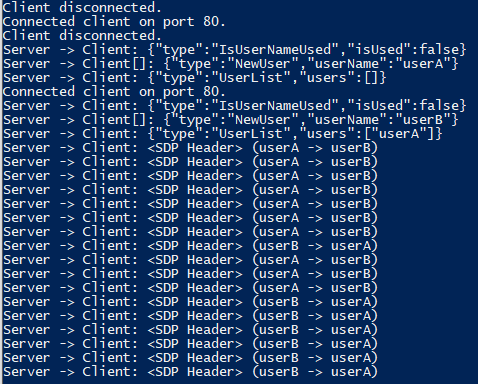
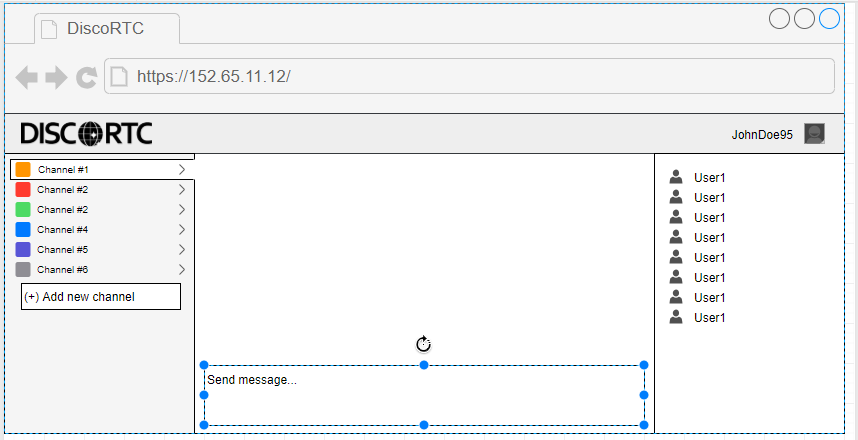


Figure 30. Két kliens távozása, majd két új kliens csatlakozása és SDP cseréje

#### Kliens

Az első kliens terv az alábbi volt és ez a későbbiekben nem is változott. Ragaszkodtam hogy a bal oldalsávon szobákat és usereket jeleníthessek meg.



A terv szerint első közelítésben mindenki egyedi felhasználóként beszélhet a másikkal, tehát csatornák létrehozására nincs lehetőség. Miután kiválasztunk egy aktív felhasználót beszélgetéshez, opcionálisan behívhatunk a csoportba másokat is. Mivel ez az implementáció nem használja fel az SFU/MCU nyújtotta lehetőségeket, ezért minél több embert hívunk be a beszélgetésbe, annál jobban nő az adatforgalom, ami már 3 emberrel való beszélgetés esetén is leterhel egy egyszerű számítógépet.

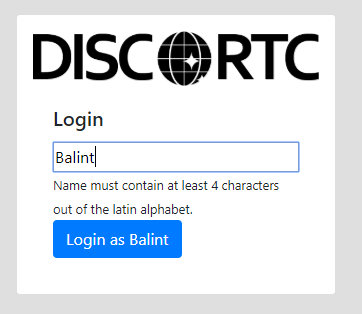


Figure 31. A bejelentkező képernyő

Kliens oldalon az Angular 5 framework-öt használtam, így nem is volt lehetőség dönteni, hogy TypesScript-ben, vagy anélkül fogjak neki a kódoláshoz. Szerencsére a kezdeti nehézségek később megtérültek.

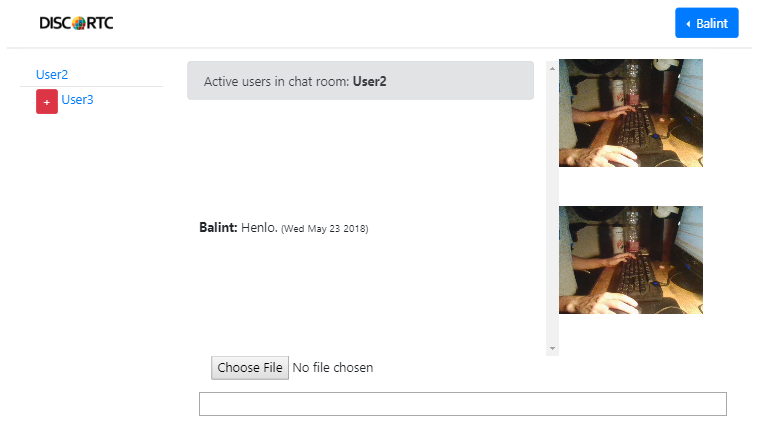


Figure 32. Chatelés egy klienssel

Két oldal van, a login és a chat. A login oldalon egyedi felhasználónevet ellenőrzi a kliens, majd regisztrál minket az aktív chat-elők között, míg a chat oldalon a felhasználókat választhatjuk ki és beszélgethetünk velük. Miután a felhasználók listájából kiválasztunk egy embert, felépül közöttük a WebRTC kapcsolat. Ezután ha szeretnénk mégtöbb embert behívni a szobába, a listában lévő piros pluszjelre kell kattintani. Ilyenkor egyrészt nő a video stream-ek száma, másrészt a felső szürke mezőben megjelenik az új résztvevő neve.

Ahogy azt az Angular-nél megszokhattunk, az üzleti logikát service-ek implementálják. Itt sincs másképp, egy service kezeli a WebRTC kapcsolatokat, illetve egy service kezeli a WebSocket kapcsolatot. A WebSocket-et a socket.io-client könyvtár kezeli, míg a WebRTC kapcsolatot a simple-peer.

A simple-peer könyvtár, ahogy azt már említettem, egy kicsit segít enyhíteni a WebRTC kapcsolat komplexitásán, de mivel a signaling-ot és a stream-ek kezelését nekünk kell megoldanunk, a kapcsolat inicializálása ugyanolyan komplikált maradt, mint a proof of concept esetén.

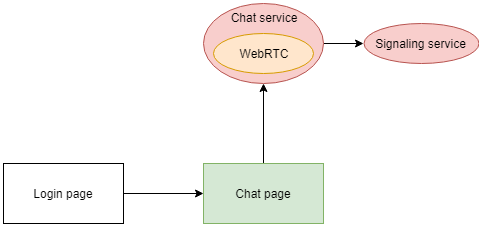


Figure 33. Az első webapp verzió architektúrája.

#### Összegzés

A WebRTC hátránya, hogy nagyon sok esemény keletkezik és dolgozódik fel egy event handler-ben. Ezeket pedig összefogni nagyon átgondolt struktúrát követel meg. Az első verzió elvetésének oka nem is a szerver hiányosságai, hanem inkább a kliens oldal kezelhetetlensége, annak továbbfejlesztésének reménytelensége.

### Második verzió

#### Szerver

Újraterveztem a teljes backend architektúrát. Először is az általam – szintén – jól ismert C# nyelvre váltottam, illetve komolyabban felülvizsgáltam, hogy mi kellhet szerver oldalon. Először is több különálló szervert hoztam létre.

Mivel a signaling teljesen különálló szolgáltatás, egyedül a kliens érkezése és távozása esetén hajtódik itt végre domain specifikus esemény. Ez ok arra, hogy az SDP cserét különválasszam például a kliensek listázásától. A két szerver teljesen állapot mentes abból a szempontból, hogy horizontális skálázás esetén, vagy a szerver újraindulása esetén nem veszne adat, nem kerülne inkonzisztens állapotba a backend oldal. (Leszámítva a WebSocket framework-öt, mivel a SignalR eltárolja az adott kliens adatait azonosításhoz.)

Mivel a SignalR framework-öt választottam a websocket szerverhez, ezért néhány szót szólnom kell róla. Alapból ugyanaz a technológia mint más websocket szerver implementációk, de egyrészt a Microsoft által támogatott, így végiggondolt és valószínűleg stabil eszköz, másrészt sokkal egyszerűbbé teszi a szolgáltatások bővítését.



Figure 34. Szerver (hub) és Böngésző (proxy) kétirányú kommunikációja

Egy egyszerű ASP.NET Core projektre épül a SignalR, ahol miután a service-ekhez és az app-hoz adtuk, már csak mappelni kell egy adott route-ra a létrehozott hub-ot. A következő kódrészlet pont ezt teszi:

app.UseSignalR(routes =>

{

routes.MapHub<SignalServerHub>("/chat");

});

Később, amikor a böngészőből csatlakozunk, a “chat” nevű route-ot kell megadni. Ez lesz a hub, a SignalServerHub pedig a hub implementáció. Gondolhatunk erre úgy, mint egy chat szobákat összefogó szolgáltatás lista. Ezt nekünk kell létrehozni a SignalR.Hub osztályból való leszármazással. Az ezután megírt metódusok a különböző események lesznek, amiket elérhetünk kliens oldalról.

Mivel lokálisan több porton is kommunikál a klienssel a két szerver, ezért fontos, hogy a CORS-nek megfeleljen a szerver által küldött header.

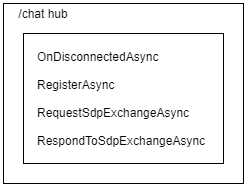


Figure 35. A hub összetétele.

Ahogy azt az előbbi ábrán (Figure 35) láthattuk, csak a legszükségesebb eseményeket kezeli le a signaling szerver, minden más a http api szolgál ki. Az OnDisconnected értesíti a többi klienst az aktuális kliens távozásáról. A Register regisztrálja az újonnan érkező klienst. Bár ezzel egy probléma van, hogy nincs köze az SDP cseréhez. Igaz, az OnDisconnected-et is le lehetne cserélni egy egyszerű API hívásra „onunload” esemény érkezésekor a böngészőben. De ez a jelenlegi viselkedést nem befolyásolja. A RequestSdpExchange továbbítja a hoszt kérését a vendégnek, aki fogadja a kérést. Nagyon praktikus lett volna, de sajnos nincs lehetőség arra, hogy a kliens értéket adjon vissza a szervernek, ezért miután a hoszt átpasszolta az SDP header-t a vendégnek, muszáj egy külön hívást intézni annak a RespondToSdpExchange felé, hogy a hoszt megkapja az SDP answer-jét. (A kliens leírásánál részletezem, hogyan lehet használni a SignalR proxy-t a böngészőben)

A userekről az adatokat egy Redis cache tárolja, így végre perzisztens az információ róluk. Miután az ASP.NET applikáció inicializálásánál felvettem a ChatCacheService osztályt, a hub implementációm dependency injection-nel megkaphatja azt inicializálva a cache nevével, amit beállítottam, és felhasználhatja a felhasználók adatainak kezelésére. Mivel a Redis csak primitív típusokat tud tárolni, ezért kulcsként jelenleg a felhasználó nevét használom, mint egyedi azonosító, értéknek pedig egy json-ként sorosított objektumot, ami egy user entitás típus, névvel és kapcsolat ID-val, ami a hub implementációban ki tudja választani az adott felhasználót, hogy direkt üzenetet küldjön neki.

Az API jelenleg egy szolgáltatást nyújt, visszaadja az aktív userek listáját. Jobb esetben itt lenne még a regisztráció és a kilépés. Amennyiben a chat program szolgáltatása kibővül, ezeknek az endpoint-oknak a száma ugrásszerűen megnő, hiszen egyre több információt kell számon tartani a felhasználók közti interakcióról.

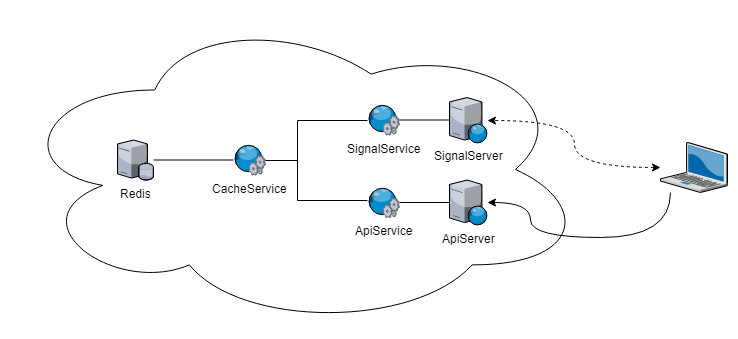


Figure 36. Robosztusabb szerver architektúra

Összefoglalva a szerver robosztusabb lett, könnyebben módosítható. Tervben van, hogy docker image-be rakjam a kész szervereket és így tetszőleges felhő szolgáltatásba deploy-olható legyen.

#### Kliens

A kliens az előző változathoz hasonlóan, bár egy verziót ugorva Angular 6-ban lett megírva. Ezúttal nem foglalkoztam a megjelenítéssel, csak azt a célt tűztem ki magam elé, hogy olyan klienst készítsek, ami megbízhatóan működik, újra felhasználható, bármikor bővíthető és az eddigi problémákat orvosolja. Szerencsére ezeket a gondokat valóban sikerült orvosolnom, illetve egyéb elegáns megoldásokkal bővítettem a kliens oldalt.

Hasznos képessége az Angularnak az environment.ts fájl. Ahogy Asp.Net-ben az appsettings.json tárolja a konfigurációt, ugyanúgy itt ez a fájl. Struktúráltan tárolhatom a különböző url-eket, globális beállításokat, stb.

A komponenseket Angular-ban kétfelé csoportosítottam. A page mappában csak a teljes ablakot kitöltő komponensek, önálló route-ok foglalnak helyet, míg a shared mappában minden más, így a sidebar, messagebox és messagebox container is, melyek a jelenlegi interfész építőkövei. Két route van, egyrészt a default, a login ablak, illetve a második, a chat ablak amit már az előző verzióból megismertünk. Mivel az egyes oldalak mérete nőhet az idő előrehaladtával, ezért már az elején lazy loading-gal leválasztottam a két aloldalt a főoldaltól. Így ha később csak a chat ablakba lépünk be, nem kell letölteni minden erőforrást a login oldalhoz.

Különböző osztályokat kellett létrehoznom az entitások tárolásához. Egyrészt a UserEntity-t, amit már Redis-ben is tároltunk szerver oldalon, ezen kívül szükség volt a DataChannel létrehozásához egy tárolóra, illetve egy újabb osztály a chat üzenetek fogadásához.

Angularban a legfontosabb entitás a komponenseken kívül a service. Ez a réteg hivatott az üzleti logikát megvalósítani. A webappban a következő service-ek működnek:

***ApiService***

Az ApiService egy segéd service, abban az értelemben, hogy teljesen általános http kliens implementációt rejt. Mivel generikus, ezért bármilyen objektummal vissza tud térni. Az Angular-nek is van egy beépített http kliense, ám az sokkal kevesebb lehetőséget nyújt, bizonyos header információkat nem tölt ki, így saját wrappert kellett köré írnom.

GlobalService

A GlobalService a globális változókat tárolja melyet így mindenhonnan el lehet érni, ahol dependency injection-nel megkapunk a konstruktorban.

UserInterfaceService

A UserInterfaceService feladata, hogy a böngészőablakban történő grafikus interfész specifikus eseményeket lekezelje. Jelenleg két eseményre lehet benne feliratkozni, a felhasználó kiválasztásra, illetve az új üzenet érkezésére. Ezeknek az eseményeknek a service-en keresztül nagy haszna van, mivel sokszor a program nem tudja azonnal érvényre juttatni a változásokat. Az események bekövetkezésekor teljesen asszinkron módon megváltozik egy komponens belső állapota, az Angular változás detektáló motorja pedig amikor éppen ráér, ezt a változást megjeleníti.

LoginService

A LoginService jelen állapotában két funkciót lát el. Egyrészt létrehozza a session-t, azaz a signaling service segítségével regisztrál, másrészt pedig átnavigálja az aktuális oldalt a /chat aloldalra.

SessionService

A SessionService feladata, hogy a megadott user privilégiumok alapján beregisztrálja a felhasználót. Jelenleg csak egyirányú kommunikáció zajlik, azaz a signaling szerver kötelezően elfogadja a regisztrációt, ám később be lehet vezetni az authentikációt.

SignalingService

A SignalingService kifelé egy nagyon egyszerű API-t nyújt, ám a belső felépítése magyarázatra szorul. Magját egy tagváltozó, a HubConnection adja. Ez az objektum felel a SignalR proxy működéséért, a SignalR SDK része. A service inicializálásánál ez az objektum épül fel, hogy tudja fogadni a kéréseket. Mielőtt rátérek az osztály részére, meg kell említenem, hogy a SignalR kétféle hívást különböztet meg. Vannak action-ök és a function-ök. Az action nem tér vissza értékkel, míg az utóbbi igen. Ezért volt jó döntés ez a könyvtár, mivel out of the box támogatja az RPC hívást, nem kell trükközni a Websocket API-n keresztül. Tehát a SignalingService egyrészt áll magából a hub connection-ből, másrészt kétfajta metódusból. Az egyik fajta csak egy delegate, ami azt mondja meg, hogy a szerver felől milyen paraméterekkel milyen függvény hívás érkezhet. Ezek a függvények önmagukban nem képesek tenni semmit. Rájuk a subscribe nevű property-n keresztül lehet feliratkozni, ami egy Proxy, azaz kiajánlja a SignalingService property-jeit és amikor valamilyen műveletet akarunk végezni rajtuk, akkor ezt a Proxy elkapja és mi megváltoztathatjuk a viselkedését. Egy rövid példán keresztül szemléltetem:

this.signalingService.subscribe.OnUserConnected = this.addNewUser;

this.signalingService.subscribe.OnUserDisconnected = this.removeUser;

Tehát event handler stub-okra iratkozhatunk fel, ami valójában a HubConnection eseménye.

A másik hasonlóan kényelmes, de nehezebben implementálható rész az action és a function. E kettő is csak stub, de őket nem egy Proxy kezeli, hanem a rájuk rakott dekorátor.

@RemoteAction()

public async RequestSdpExchangeAsync(guestId: string, hostSdpHeader: RTCSessionDescriptionInit) {}

Amikor meghívjuk ezeket a függvényeket, a dekorátor áthív a HubConnection send és invoke metódusába, és annak eredményével is tér vissza. Ezzel a két módszerrel megoldottam, hogy a SignalingService tetszőlegesen bővíthető legyen kódismétlés nélkül.

UserService

A UserService egyetlen feladata jelenleg, hogy az ApiService segítségével lekérdezze az ApiServer-től az online klienseket.

WebRtcService

A WebRtcService hasonlóan bonyolult, mint a SignalingService, ám ez nem a kifelé mutatott eleganciához szükséges, hanem a WebRTC eseményeinek kordában tartásához. Sok helyen kell feliratkozni rájuk melyek logikailag és a kódban is igen távol vannak egymástól. A következő szekvencia diagram bemutatja, hogy service szinten milyen hívások szükségesek a kapcsolat létrehozásához.

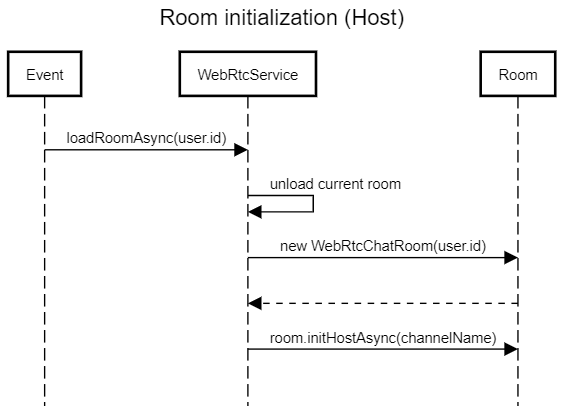


Figure 37. Szoba létrehozása.

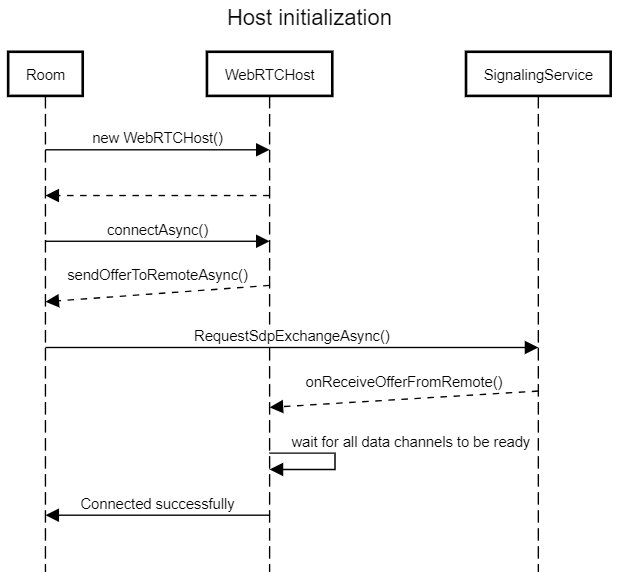


Figure 38. WebRTCHost kapcsolódása.

Amikor az összes data channel készen áll és megérkezett a videó/audio stream, a connectAsync függvény visszatér.

Ezen kívül már csak a utils mappa van, ahol egyrészt a SignalingService dekorátorai foglalnak helyet, illetve a WebRtcService low-level részei, amik hasonlóan a proof of concept-hez, a böngésző apit hívogatják.

# Elkészült rendszer értékelése

## Ellenőrzés

## Teljesítmény

## Szerzett tapasztalatok

# Összefoglaló

# Függelék