



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Távközlési és Mesterséges Intelligencia Tanszék

Nézők által konfigurálható elő videóközvetítés Media over QUIC protokollal

SZAKDOLGOZAT

Készítette
Schweitzer András Attila

Konzulens
Németh Felicián

2025. december 7.

Tartalomjegyzék

Kivonat	i
Abstract	ii
1. Bevezetés	1
1.1. A téma aktualitása és a tartalomelőállítás kiszélesedése	1
1.2. A megoldás koncepciója és Célkitűzés	2
1.3. A dolgozat felépítése	2
2. Technológiai háttér	3
2.1. A QUIC protokoll szerepe	3
2.2. Media over QUIC (MoQ) architektúra	3
2.2.1. Hálózati szereplők és a Relay modell	4
2.2.2. MoQ Adatmodell és Objektum hierarchia	4
2.2.3. A katalógus (Catalog) szerepe és felépítése	4
2.2.4. Publikációs és Feliratkozási mechanizmusok	6
2.3. A CMASF formátum szerepe az alacsony késleltetésben	6
2.4. Videó tartalom leképezése MoQ objektumokra	7
2.5. FFmpeg multimédia keretrendszer	7
2.6. Elérhető MoQ implementációk	7
3. Követelményelemzés és Rendszerspecifikáció	9
3.1. Funkcionális követelmények	9
3.2. Nem funkcionális követelmények	10
3.3. A rendszer határai (Scope)	10
3.4. Sematikus rendszerkoncepció	10
3.4.1. Architekturális szerepkörök	10
3.4.2. Működési logika és Információáramlás	11
4. Tervezési kérdések és döntések	13
4.1. Skálázhatóság: Átkódolók száma egy média névtérben	13
4.2. Működési hatókör: Egyetlen forrás-névtér elve	14
4.3. Erőforrás-hatékonyság: Privát vagy Nyilvános sávok?	14
4.3.1. A „Privát sáv” megközelítés	15
4.3.2. A „Nyilvános sáv” és a cache-hatékonyság	15
4.4. Dinamikus metaadatok: A Katalógus kezelése	15
4.4.1. Lehetséges stratégiák	15
4.4.2. A választott megoldás: Katalógus Készítő (Catalog maker)	16
4.5. A felhasználói igények jelzése: Implicit vs. Explicit modell	16
4.5.1. Az implicit, névtér alapú megközelítés korlátai	16
4.5.2. A választott megoldás: Explicit JSON alapú kérés	17

4.6. A válaszadás aszimmetriája és az explicit elutasítás hiánya	17
4.7. Tartalmak szinkronizációja és azonosítása	18
5. Rendszerterv és protokoll kialakítás	19
5.1. Névtér és sávstruktúra tervezése	19
5.1.1. Kezdeti névtérkoncepció	19
5.1.2. Végeleges névtér-kialakítás	20
5.1.3. Kimeneti névtér és determinisztikus sávnevek	21
5.2. A kérés adatmodelljének gyakorlati terve	22
5.2.1. JSON formátum választása	22
5.2.2. Adatmodell felépítése	22
5.3. Rendszerkomponensek és a kommunikációs folyamat közöttük	23
5.3.1. Szerepkörök az architektúrában	24
5.3.2. Részletes kommunikációs folyamat	24
6. Implementáció	29
6.1. Fejlesztési környezet	29
6.2. Kiindulási állapot	29
6.3. A Catalog Maker megvalósítása	29
6.3.1. Belső architektúra és adatmodell	29
6.3.2. Eseményvezérelt működési logika	30
6.3.3. Dinamikus felfedezés implementációja	30
6.4. A Transcode kliens belső logikája	30
6.4.1. Aszinkron feladatkezelés	30
6.4.2. Média processzálás és adatútvonal	31
6.5. A kéréskezelő logika és a kliens működése	31
7. Kiértékelés	33
7.1. Tesztkörnyezet és konfiguráció	33
7.2. Funkcionális ellenőrzés	33
7.3. Teljesítménymérések	33
7.3.1. Késleltetés (Latency)	34
7.3.2. Erőforrás-igény és Pufferkezelés	34
7.4. Tesztek összegzése	35
8. Összefoglalás és további fejlesztési irányok	36
8.1. Munka összefoglalása	36
8.2. Továbbfejlesztési lehetőségek	36
8.2.1. Protokoll-szintű mélyítés és MoQ funkciók	37
8.2.2. Az átkódoló komponens optimalizálása	37
8.2.3. Új átkódolási funkciók és szolgáltatások	37
8.2.4. Skálázhatóság és Orkesztráció	37
Köszönhetetlenítás	38
Irodalomjegyzék	39
Függelék	41
F.1. Projekt GitHub repozitórium	41
F.2. Request JSON séma	41

HALLGATÓI NYILATKOZAT

Alulírott *Schweitzer András Attila*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot 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(k), 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 autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervezet esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2025. december 7.

Schweitzer András Attila
hallgató

Kivonat

A szakdolgozat a videóközvetítési technológiák legújabb irányvonalát, a Media over QUIC (MoQ) protokollt és annak gyakorlati alkalmazhatóságát vizsgálja. A munka célja egy olyan közvetítő rendszer megvalósítása volt, amely szakít a hagyományos, szerveroldali statikus konfigurációkkal, és lehetővé teszi a nézők számára, hogy közvetlenül befolyásolják a tartalom paramétereit. A dolgozat bemutatja a MoQ protokoll architektúráját, kiemelve azokat a szállítási rétegbeli előnyöket – mint az objektum-alapú multiplexelés és a korszerű torlódáskezelés –, amelyek alkalmassá teszik az interaktív médiaalkalmazások kiszolgálására.

A tervezés során egy elosztott architektúra került kialakításra, amelyben a vezérlés a tartalomfogyasztó kezébe kerül. A rendszer egy saját definíciójú, JSON formátumú jelzési protokollt alkalmaz, amelyen keresztül a kliensek explicit átkódolási igényeket fogalmazhatnak meg. Ez a megoldás biztosítja, hogy az erőforrás-igényes transzkódolási folyamatok kizárálag valós felhasználói igény esetén induljanak el, optimalizálva ezzel a szerveroldali terhelést. A rendszer logikai felépítése elkülöníti a tartalmak adminisztrációját és a források egységesítését végző Katalógus Készítőt, a számításigényes médiafeldolgozást végző Átkódoló klienst, valamint a felhasználói interakciót kezelő klienseket.

A rendszer implementációja C++ környezetben, a szabványtervezetet követő LibQuicR könyvtár és az FFmpeg keretrendszer integrációjával valósult meg. A fejlesztés során a fő kihívást a hálózati eseménykezelés és a valós idejű médiafeldolgozás aszinkron összehangolása jelentette. A megvalósított szoftvermodulok biztosítják a folyamatos adatcsatornát az eredeti forrás, az átkódoló és a végfelhasználó között. A megoldás modularitása lehetővé teszi a komponensek jövőbeli független skálázását és a rendszer funkcionális bővítését.

A rendszer működőképességének igazolása élő közvetítés emulálásával valósult meg. A funkcionális tesztek megerősítették, hogy a rendszer képes az emulált közvetítést dinamikusan, a kliens által definiált paraméterek szerint átkódolni és publikálni. A teljesítménymérések során vizsgált reakcióidő – a kérés elküldése és az első képkocka megjelenése közötti időtartam –, valamint a teljes lánon mért késleltetés igazolta, hogy a megoldás az élő közvetítések késleltetési időskáláján marad. A kutatás eredményei, valamint a demonstrációhoz biztosított forráskód és szkriptek alapot szolgáltatnak a MoQ protokoll további gyakorlati kutatásához.

Abstract

This thesis examines the latest direction in video streaming technologies, the Media over QUIC (MoQ) protocol, and its practical applicability. The aim of the work was to implement a delivery system that departs from traditional static server-side configurations and enables viewers to directly influence content parameters. The thesis presents the architecture of the MoQ protocol, highlighting transport layer benefits—such as object-based multiplexing and modern congestion control—that make it suitable for serving interactive media applications.

During the design phase, a distributed architecture was developed in which control is placed in the hands of the content consumer. The system employs a custom-defined, JSON-format signaling protocol through which clients can formulate explicit transcoding requests. This solution ensures that resource-intensive transcoding processes are initiated only upon actual user demand, thereby optimizing server-side load. The logical structure of the system separates the Catalog Maker, responsible for content administration and source unification; the Transcoder client, performing computation-intensive media processing; and the clients handling user interaction.

The system implementation was realized in a C++ environment through the integration of the LibQuicR library, which follows the draft standard, and the FFmpeg framework. During development, the main challenge was the asynchronous coordination of network event handling and real-time media processing. The implemented software modules ensure a continuous data channel between the original source, the transcoder, and the end-user. The modularity of the solution allows for future independent scaling of components and functional expansion of the system.

Verification of the system's functionality was achieved by emulating live streaming. Functional tests confirmed that the system is capable of dynamically transcoding and publishing the emulated stream according to parameters defined by the client. Reaction time examined during performance measurements—the interval between sending the request and the appearance of the first frame—as well as the latency measured across the entire chain, verified that the solution remains within the latency timeframe of live broadcasts. The research results, along with the source code and scripts provided for the demonstration, provide a basis for further practical research into the MoQ protocol.

1. fejezet

Bevezetés

1.1. A téma aktualitása és a tartalomelőállítás kiszélesedése

Az élő videóközvetítés (live streaming) napjainkra az internetes forgalom egyik legdominantabb szegmensévé vált. Míg korábban ez kizárolag nagy TV társaságok kiváltsága volt, mára a tartalomelőállítás kiszélesedett: bárki, aki rendelkezik okostelefonnal vagy egyéb számítógéppel, potenciális műsorszóróvá válhat. A szabványosítás alatt álló Media over QUIC (MoQ) protokoll ígérete, hogy ezt a tömeges tartalommegosztást alacsony késleltetés mellett teszi lehetővé nagyszámú felhasználó számára.

Problémafelvetés: A végfelhasználói eszközök korlátai

Bár a közvetítés lehetősége adott, a minőségi kiszolgálás technikai akadályokba ütközik. A professzionális streaming szolgáltatók (pl. Netflix, YouTube) szerveroldalon, vagy drága célhardverekkel állítják elő a videó több különböző minőségű változatát (Adaptive Bitrate Streaming - ABR), hogy minden néző a saját internetkapcsolatának megfelelő folyamot kaphassa.

Ezzel szemben egy átlagos tartalomelőállító eszközei – legyen szó egy középkategóriás laptopról, egy mobiltelefonról de akár egy felső kategóriás asztali számítógép – jelentősen szűkebb erőforrásokkal rendelkeznek, ami alapvetően két kritikus téren korlátozza a tartalompublikálás rugalmasságát.

Elsőként a **számítási kapacitás** és **energiahatékonyság** jelent fizikai korlátot. Mivel a videótömörítés rendkívül számításigényes feladat, egy mobileszköz számára már egyetlen jó minőségű videófolyam valós idejű kódolása is jelentős processzorterhelést és gyors akkumulátor-merülést eredményez. Ebben a kontextusban több párhuzamos variáns (pl. 1080p, 720p, 480p) egyidejű előállítása nehezen lehetséges az eszköz korlátozott erőforrásaival, vagy olyan kompromisszumokkal járhat, amely más téren jelentős hátrányt eredményez, pl. képminőség vagy késleltetés. Emellett bár egy erőteljesebb asztali gép képes lehet több verzió előállítására könnyedén, de ha ezt egy erőforrásigényesebb játék közvetítése közben teszi, akkor a CPU és GPU erőforrások megosztása miatt a játékélmény is romolhat.

Szorosan kapcsolódik ehhez a technikai korláthoz a **feltöltési sávszélesség (uplink)** szűkössége is. A lakossági internetcsomagok és mobiladat-kapcsolatok jellemzően aszimmetrikusak, ahol a feltöltési sebesség töredéke a letöltésinek. Így még abban az elméleti esetben is, ha a forráseszköz hardveresen képes lenne több verziót előállítani, a rendelkezésre álló hálózati sávszélesség fizikailag nem tenné lehetővé ezek szimultán továbbítását a szerver felé.

Ezen hardveres és hálózati korlátok eredőjeként a forrás jellemzően kénytelen egyetlen, kompromisszumos (vagy épp a számára elérhető legjobb) minőségben közzétenni a tartalmat. Ez a kényszerű megoldás azonban szükségszerűen kizárja a közvetítésből azokat a tartalomfogyasztókat, akik kisebb hálózati sávszélességgel rendelkeznek, vagy az eszközük teljesítménykorlátai miatt nem képesek folyamatosan lejátszani a magasabb bitrátójú videókat.

1.2. A megoldás koncepciója és Célkitűzés

A fent vázolt erőforrás-korlátokra a hálózatba integrált intelligencia adhat választ. A dolgozatban bemutatott megoldás alapkoncepciója, hogy a tartalom adaptációjának terhét levesszük a forrásról, és azt a Media over QUIC (MoQ) továbbítóhálózat egy dedikált elemére, az átkódoló kliensre delegáljuk.

A célkitűzésem egy olyan közvetítő architektúra tervezése és implementálása, amely **igényvezérelt módon** működik. Ellentétben a hagyományos műsorszóró rendszerekkel, ahol a szerverek előre elkészítik az összes lehetséges minőség-variánst (pazarolva ezzel az erőforrásokat), a javasolt rendszerben a transzkódolás csak akkor és olyan formátumban indul el, amikor arra a hálózat felől valós nézői igény (Request) érkezik.

Kiemelt szempont a rendszer tervezésekor a transzparencia és a protokoll-szintű kompatibilitás. A megoldás nem igényel speciális vezérlőcsatornát: a kommunikáció a Forrás (Publisher), az Átkódoló (Transcoder) és a Néző (Subscriber) között tisztán a MoQ protokoll objektummodelljére és feliratkozási mechanizmusaira épül. Ez biztosítja, hogy a rendszer illeszkedjen a létező MoQ infrastruktúrába, megőrizve annak skálázhatóságát és lehetőleg minél alacsonyabb késleltetését, miközben az eredeti tartalomelőállító számára a folyamat teljesen láthatatlan marad.

1.3. A dolgozat felépítése

A szakdolgozat a fenti célkitűzések mentén, a technológiai alapoktól a konkrét implementációig vezeti végig az olvasót.

A **második fejezet** a szükséges technológiai hátteret ismerteti. Bemutatom a QUIC szállítási réteg előnyeit, valamint az erre épülő Media over QUIC (MoQ) szabvány működését, különös tekintettel a hálózati szereplők szerepköreire, az objektummodellre és a katalógus-kezelésre.

A **harmadik fejezetben** definiálom a rendszerrel szemben támasztott funkcionális és nem funkcionális követelményeket, kijelölve a tervezés határait.

A **negyedik fejezet** a tervezési döntéseket tárgyalja. Itt elemzem az architektúra kialakításának kérdéseit, a szükséges kommunikációs folyamatokat, valamint az átkódolt tartalmak és az eredeti forrás közötti szinkronizáció szükségességét.

A **ötödik fejezet** tartalmazza a részletes rendszertervet, a névterek struktúráját és logikáját és a kommunikációs protokoll definícióját. Bemutatom a komponensek közötti üzenetváltási szekvenciákat, a katalógus delta-frissítési mechanizmusát, valamint a kérések (Requests) kezelésének folyamatát a MoQ hálózaton belül.

A **hatodik fejezet** a megvalósítás részleteire tér ki, ismertetve a fejlesztés során alkalmazott eszközöket, a Catalog Maker és a Transcoder komponensek belső felépítését.

A **hetedik fejezet** a rendszer validálását és a mérési eredményeket foglalja össze. Demonstrálom a működést valós környezetben, valamint vizsgálom a megoldás által bevezetett járulékos késleltetést és az erőforrás-igény alakulását.

Végezetül a **nyolcadik fejezetben** összegzem az eredményeket, és javaslatot teszek a rendszer lehetséges továbbfejlesztési irányaira.

2. fejezet

Technológiai háttér

Jelen fejezet célja, hogy ismertesse a dolgozat alapját képező hálózati technológiákat. Rövid áttekintést adok a QUIC szállítási protokollról, majd részletesen bemutatom az IETF által szabványosítás alatt álló Media over QUIC (MoQ) architektúrát, annak adatmodelljét, üzenetkezelési mechanizmusait, valamint a fejlesztéshez kiválasztott implementációt.

2.1. A QUIC protokoll szerepe

A modern internetes videótovábbítás (HTTP/TCP alapú HLS/DASH) egyik legnagyobb gátja a TCP protokoll merevsége, különösen a *Head-of-Line Blocking* jelenség. TCP esetén, ha egy csomag elveszik, a vevőoldali operációs rendszer visszatartja az összes utána következő, már megérkezett csomagot is, amíg az elveszett adat újra meg nem érkezik. Elő videó esetén ez elfogadhatatlan késleltetést (latency) és a lejátszás megakadását (buffering) okozza.

A QUIC egy UDP-re épülő, biztonságos, megbízható szállítási protokoll[7], amely megoldja ezt a problémát. A QUIC legfontosabb tulajdonsága a videóátvitel szempontjából a **stream-ek független multiplexelése**. Egyetlen QUIC kapcsolaton belül több adatfolyam (stream) futhat párhuzamosan. Ha az egyik streamben csomagvesztés történik, az nem blokkolja a többi stream feldolgozását. Ez lehetővé teszi, hogy a videó, az audió és a vezérlőadatok egymástól függetlenül, a lehető legalacsonyabb késleltetéssel érkezzenek meg.

2.2. Media over QUIC (MoQ) architektúra

A Media over QUIC (MoQ) egy új generációs, alkalmazás rétegbeli média-elosztási protokoll, amelyet az IETF MoQ munkacsoportja fejleszt[13]. A protokoll alapvető célkitűzése, hogy betöltsse a technológiai úrt a valós idejű, interaktív kommunikáció és a tömeges műsorszórás között.

Míg a QUIC önmagában „csak” egy hatékony szállítási csatornát biztosít, a MoQ erre építve egy teljes körű továbbítási logikát definiál. A protokoll szakít a hagyományos kliens-szerver kapcsolattal, és helyette egy **Publish/Subscribe (Pub/Sub)** alapú modellt vezet be. Ennek lényege, hogy a médiaátvitel nem pont-pont kapcsolatokon, hanem egy intelligens elosztóhálózaton keresztül történik. A MoQ lehetővé teszi, hogy a tartalom elnevezett objektumok formájában, közbenső csomópontok (Relay-ek) láncolatán keresztül jusson el a forrástól a nézőig. Ez az architektúra biztosítja, hogy a rendszer egyszerre legyen képes az élő beszélgetésekhez szükséges alacsony (másodperc alatti) késleltetésre és a több millió nézőt kiszolgáló skálázhatóságra.

2.2.1. Hálózati szereplők és a Relay modell

A MoQ architektúra alapvetően három fő entitásra épül, amelyek egy logikai hálózatot alkotnak:

- **Publikáló (Publisher):** A tartalom forrása. Létrehozza a média objektumokat, és elérhetővé teszi azokat a hálózaton.
- **Feliratkozó (Subscriber):** A tartalom fogyasztója. Igényt (subscription) nyújt be adott tartalmakra, és fogadja az adatfolyamot.
- **Továbbító (Relay):** A MoQ hálózat intelligens csomópontja. Ellentétben a hagyományos IP routerekkel, a továbbító ismeri a MoQ objektummodellt, de a **tartalmába nem lát bele**. Nemcsak továbbítja a csomagokat, hanem gyorsítótárazza (cache) is azokat, és deduplikálja a kéréseket. Ha tíz néző ugyanazt a videót kéri, a továbbító csak egyszer kéri el a forrástól (vagy az előző továbbítótól), majd szétosztja azt a tíz nézőnek. Ez a mechanizmus teszi lehetővé a CDN (Content Delivery Network) szintjén skálázódást.

2.2.2. MoQ Adatmodell és Objektum hierarchy

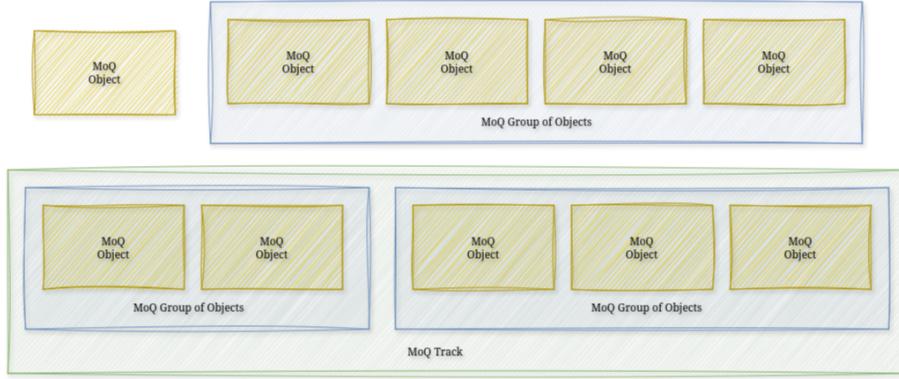
A MoQ nem byte-streamként, hanem strukturált objektumokként kezeli a médiát. Ez a hierarchy elengedhetetlen a helyes továbbításhoz és a prioritások kezeléséhez. A MoQ objektum modelljét a névtér ábrázolás kivételével a 2.1 ábra szemlélteti.

1. **Névtér (Namespace):** A legfelső szint, amely egy teljes közvetítést vagy szolgáltatást azonosít (pl. meeting-123), de akár egy körvetítésen belül lehetnek alnévterek (pl. meeting-123/room-1).
2. **Sáv (Track):** Egy logikailag összetartozó, folyamatos médiafolyam a névtéren belül. Például külön sáv a videó, az audió, vagy a jelen szempontjából fontos *katalógus*. A sáv a feliratkozás (subscription) alapegysége.
3. **Csoport (Group):** A sávon belüli időbeli egység, folyamatosan növekvő számmal azonosított. Jellemzően függetlenül dekódolható (pl. egy GOP - Group of Pictures a videótömörítésben). A továbbítók gyakran csoport szinten döntik el, hogy mit dobjanak el torlódás esetén.
4. **Objektum (Object):** A legkisebb egység (pl. egy videó képkockája). Ez kerül ténylegesen átvitelre a QUIC stream-ekben vagy datagramokban. Egy csoportnak a része, amelyen belül folyamatosan növekvő számok azonosítják a különböző objektumokat.

2.2.3. A katalógus (Catalog) szerepe és felépítése

A MoQ egyik legfontosabb specifikuma a katalógus. Ennek szükségeségét nemcsak a tartalom strukturálása, hanem a protokoll *Announce* (Hirdetés) mechanizmusának korlátai is indokolják.

A MoQ szabványban a hirdetés kizárálag **Névtér (Namespace)** hatókörben (scope) értelmezett. Ez azt jelenti, hogy a Publikáló a hálózaton keresztül csak azt képes jelezni, hogy egy adott szolgáltatás (pl. meeting-123) elérhető nála, de protokoll szinten nincs lehetőség az ezen belül lévő Sávok (Track-ek) automatikus felsorolására vagy meghirdetésére. A Feliratkozáshoz (Subscribe) azonban a kliensnek pontosan ismernie kell a sáv nevét (pl. video_1080p). Mivel ez az információ tisztán MoQ transzport szinten nem szerezhető



2.1. ábra. MoQ objektum modell egy sávon belül[10]

meg, szükség van egy applikációs szintű leírónak – ez a katalógus –, amely explicit módon definiálja a névtéren belül elérhető sávokat.

A draft-ietf-moq-warpspecifikáció alapján a katalógus egy JSON alapú dokumentum [8]. Ez tartalmazza az elérhető sávok nevét, típusát (video/audio) és technikai paramétereit (codec, felbontás, bitráta).

Fontos kiegészítés, hogy a specifikáció lehetőséget ad arra is, hogy a videó dekódolásához elengedhetetlen inicializációs adatokat (Init Segment) közvetlenül a katalógusban, az adott sávhoz tartozó metaadatként (pl. Base64 kódolva) jelenítsük meg. Ez gyorsítja a lejátszás indulását, mivel a katalógus letöltésével a dekóder konfigurálásához szükséges adatok is azonnal rendelkezésre állnak.

Render Groups és Sávváltás Ahhoz, hogy a kliens tudja, mely sávok között vált hat szabadon (pl. minőségrömlás esetén átvált 1080p-ről 720p-re), a szabvány bevezeti a **Render Group** fogalmát. Azok a sávok, amelyek ugyanannak a vizuális tartalomnak az alternatív reprezentációi (és időben szinkronban vannak, azaz „time-aligned” sávok), ugyanahhoz a renderGroup-hoz tartoznak a katalógusban. A kliens bármikor válthat két olyan sáv között, amelyeknek a renderGroup azonosítója megegyezik. Példa egy egyszerű katalógus struktúrára (JSON reprezentáció):

```
{
  "tracks": [
    {
      "name": "video_1080p",
      "renderGroup": 1,
      "width": 1920,
      "height": 1080,
      "framerate": 30
    },
    {
      "name": "video_720p",
      "renderGroup": 1,
      "width": 1280,
      "height": 720,
      "framerate": 30
    },
    {
      "name": "audio_main",
      "renderGroup": 2,
    }
  ]
}
```

```

        "bitrate": 128000
    }
]
}

```

A fenti példában a kliens látja, hogy a video_1080p és video_720p ugyanahhoz a csoporthoz (renderGroup: 1) tartozik, így dinamikusan válthat közöttük, míg az audió (renderGroup: 2) egy független folyam.

Katalógus frissítés Dinamikus tartalmak esetén a katalógus frissülhet (pl. új minőség jelenik meg). A szabvány támogatja a delta frissítés (**Delta Update**) mechanizmust, ahol nem kell a teljes katalógust újra letölteni, elegendő csak a változásokat (új sáv hozzáadása/törlése) teríteni a hálózaton.

2.2.4. Publikációs és Feliratkozási mechanizmusok

A MoQ működése a Publish/Subscribe (Pub/Sub) modellre épül, amely aszinkron és igényvezérelt. A katalógus ismeretében a folyamat a következő lépésekkel áll:

- **Hirdetés (Announce):** A publikáló (vagy a nevében eljáró továbbító) jelzi a hálózatnak, hogy egy adott névtér (Namespace) elérhető nála. Ez még nem tartalmaz médiaadatot, csak az elérhetőség tényét.
- **Feliratkozás (SUBSCRIBE):** A feliratkozó (vagy egy köztes továbbító) jelzi, hogy igényt tart egy adott sáv tartalmára. A feliratkozás a sáv nevére és a névtérre hivatkozik. A továbbító hálózat a feliratkozásokat visszavezeti a forrás felé.
- **Objektum (OBJECT):** Ha van érvényes feliratkozás, a publikáló megkezdi az objektumok küldését. A továbbítók csak azon az ágon továbbítják az adatot, ahonnan érvényes feliratkozás érkezett, így minimalizálva a felesleges hálózati forgalmat.

2.3. A CMAF formátum szerepe az alacsony késleltetésben

A modern streaming megoldások, így a MoQ esetében is, kritikus kérdés a médiaadatok csomagolása. A **CMAF (Common Media Application Format)**[6][11] egy szabványosított konténerformátum, amelynek célja, hogy egységesítse a különböző streaming protokollok (pl. DASH, HLS) alatt használt médiafájlokat.

Az élő, alacsony késleltetésű (Low-Latency) átvitel szempontjából a CMAF legfontosabb tulajdonsága a **fragmentálhatóság**. A hagyományos videofájlokkal ellentétben, ahol a metaadatok és a médiaadatok nagy tömbökben helyezkednek el, a fragmentált CMAF (fCMAF) apró, önállóan is értelmezhető egységekre bontja a folyamot.

Egy CMAF folyam két fő elemből épül fel:

- **Inicializációs szegmens (Init Segment):** Ez tartalmazza a videó dekódolásához szükséges statikus adatokat (pl. codec típus, felbontás). A CMAF struktúrában ez az ftype és moov atomokból áll.
- **Média fragmentumok (Media Fragments):** Ezek hordozzák a tényleges videó- és audioadatokat. Egy fragmentum egy moof (Movie Fragment) atomból – amely a pontos időzítést és pozíciót írja le – és egymdat (Media Data) atomból áll.

Ez a struktúra teszi lehetővé, hogy a lejátszó (vagy a hálózati továbbító) már az előtt megkezdje egy videószegmens feldolgozását, hogy a teljes szegmens letöltődött volna (Chunked Transfer Encoding), ami elengedhetetlen a valós idejű átvitelhez.

2.4. Videó tartalom leképezése MoQ objektumokra

A fent bemutatott CMAF struktúra és a MoQ objektummodell összekapcsolását (specifikusan a WARP streaming formátum [8]) szigorú szabályok definiálják. Ez a leképezés biztosítja az alacsony késleltetést és a hatékony továbbítást a hálózaton keresztül.

A leképezés alapelvei a következők:

- **Sáv (Track):** Egy videófolyam egyetlen variánsa (pl. 1080p 30fps) egy önálló sávnak felel meg.
- **Csoport (Group) = GOP:** A videótömörítésben használt Group of Pictures (GOP) struktúra közvetlenül a MoQ csoportokra képeződik le. minden új GOP (amely egy kulcsképkockával, azaz keyframe-mel indul) egy új MoQ csoportot indít.
- **Objektum (Object) = frame:** Általában egy videó képkocka – vagy pontosabban egy CMAF fragmentum (moof+mdat) – egy MoQ objektumnak felel meg.
- **Szinkronizáció (keyframe = object 0):** A szabvány előírja, hogy minden csoport első objektuma (Objektum ID: 0) kötelezően egy kulcsképkocka (keyframe) legyen. Mivel ezek önmagukban, előzmény nélkül (leszámítva az inicializáló szegmenst) dekódolhatók, ez garantálja, hogy ha egy kliens egy új csoport elején kapcsolódik be, azonnal megkezdheti a lejátszást anélkül, hogy meg kellene várnia az előző csoport adatait. Ez biztosítja hogy a továbbítók úgy tudják optimalizálni a továbbítást, hogy nem tudják pontosan milyen tartalom van az objektumokban, elegendő csak a struktúra ismerete.

2.5. FFmpeg multimédia keretrendszer

Bár a MoQ a továbbításért, a CMAF pedig a média formátumért felel, a videójel manipulációja (átkódolás, méretezés) külön technológiát igényel. Az **FFmpeg** a multimédiás fájlok és folyamok rögzítésére, konvertálására és streamelésére szolgáló, ipari szabványnak tekinthető nyílt forráskódú keretrendszer[3].

A rendszer modularitása teszi alkalmassá a tervezett transzkódoló egységbe való integrálásra:

- **libavcodec:** A kodekek széles skáláját támogató könyvtár, amely lehetővé teszi a nyers videóadatok tömörítését (encoding) és kibontását (decoding).
- **libavformat:** Ez a modul felel a konténerformátumok (muxing/demuxing) kezeléséért. Képes a hálózatról érkező adatfolyamot szétszedni elemi videójelekké, majd a feldolgozás után újra érvényes konténerbe (jelen esetben CMAF-ba) csomagolni.
- **libswscale:** A képpontszintű műveletekért (pl. átméretezés, színtér-konverzió) felelős könyvtár.

Az FFmpeg képes a bemeneti adatokat folyamként (stream) kezelní, így nem szükséges teljes fájlok megléte a feldolgozás megkezdéséhez, ami elengedhetetlen az élő közvetítések átkódolásánál.

2.6. Elérhető MoQ implementációk

Mivel a MoQ egy aktívan fejlesztés alatt álló IETF szabvány, az implementációs környezet folyamatosan változik. A különböző nyelveken írt könyvtárak eltérő funkcionálitást és érettségi szintet mutatnak. Az alábbiakban a fejlesztés szempontjából legrelevánsabb projekteket ismertetem technikai képességeik alapján.

Rust: moq-rs egy Rust nyelven írt[1] és az egyik legfejlettebb nyílt forráskódú implementáció. Kiemelkedő tulajdonsága, hogy a transzport réteg mellett magas szintű médiatámogatással is rendelkezik: képes videó és audió tartalmakat a MoQ objektummodelljére leképezni és továbbítani. Ugyanakkor a funkciókészlete még nem fedi le a szükséges mértékben tervezett szabványt: jelenlegi állapotában például hiányzik belőle a SUBSCRIBE_NAMESPACE (névtér szintű feliratkozás) teljes körű támogatása, ami korlátozó tényező lehet.

Go: moqtransport egy Go nyelvű[4] könyvtár, architektúrájában a LibQuicR-hez hasonlít: elsősorban a "nyers" MoQ transzport réteg megvalósítására fókuszál. Biztosítja az alapvető objektum-küldési és fogadási primitíveket, valamint a QUIC kapcsolatok kezelését, de nem tartalmaz magas szintű média-logikát (pl. beépített videó tároló kezelést). A Go nyelvi környezet miatt kiválóan alkalmas backend mikroszolgáltatások fejlesztésére.

TypeScript / Rust: kixelated/moq [2] repozitórium (amely Rust és TypeScript komponenseket is tartalmaz) elsődleges célja a MoQ eljuttatása a böngészőbe. A TypeScript implementáció a WebTransport API-ra építve teszi lehetővé, hogy webes kliensek csatlakozzanak a MoQ hálózathoz. Ez a könyvtár kulcsfontosságú a kliensoldali megjelenítéshez, mivel lehetővé teszi a videófolyamok lejátszását HTML5 környezetben speciális bővítmények nélkül.

C++: LibQuicR projektet választottam a feladat megvalósításához[14]. Ez az implementáció a Go verzióhoz hasonlóan alapvetően egy alacsony szintű könyvtár, amely az üzenetküldést és a kapcsolatkezelést valósítja meg, de videóküldési képességgel az eredeti projekt nem rendelkezik.

A LibQuicR melletti választást az indokolta, hogy egy korábbi munkám során már kiegészítettem ezt a könyvtárat egy alapszintű videó streamelési funkcionalitással. Ez a módosítás tette képessé a LibQuicR-t arra, hogy az FFmpeg keretrendszert használva valós médiaadatokat (CMAF objektumokat) továbbítson. A meglévő kódmezők ismerete és a C++ környezet (amely megkönyíti az FFmpeg integrációt) miatt ideális alapot nyújt a feladat elvégzéséhez.

3. fejezet

Követelményelemzés és Rendszerspecifikáció

Jelen fejezet a korábban ismertetett technológiai háttér és a feladatkiírás alapján definiálja a rendszerrel szemben támasztott funkcionális és nem funkcionális követelményeket, kijelöli a fejlesztés határait, valamint bemutatja a megoldás szemantikai koncepcióját.

3.1. Funkcionális követelmények

A rendszer alapvető feladata, hogy feloldja a forrásoldali erőforrás-korlátokból adódó me-revséget, és lehetővé tegye a heterogén kliensoldali igények kiszolgálását. A tervezés során az alábbi funkcionális követelményeket határoztam meg:

- **Néző által definiált formátumkérés:** A rendszernek biztosítania kell, hogy a Feliratkozó (Subscriber) explicit jelezhesse igényét egy olyan videóformátumra (pl. alacsonyabb felbontás, eltérő bitráta), amely eredetileg nem áll rendelkezésre a publikált katalógusban. A rendszernek képesnek kell lennie ezen igények fogadására és értelmezésére.
- **Igényvezérelt működés (On-demand transcoding):** Az erőforrás-pazarlás elkerülése érdekében az átkódolási folyamat kizárolag akkor indulhat el, ha legalább egy néző aktív igényt nyújtott be az adott formátumra.
- **Forrás-agnosztikus, transzparens működés:** Kiemelt tervezési szempont, hogy az eredeti tartalomelőállító (Publisher) működését ne kelljen módosítani. A forrásnak nem kell tudnia arról, hogy a tartalmát átkódolják; ő kizárolag a szabványos MoQ hálózattal kommunikál [8]. A rendszernek transzparens módon kell beépülnie a közvetítési láncba, úgy, hogy a forrás számára láthatatlan maradjon. Ez a transzparenzia biztosítja a megoldás univerzális és dinamikus alkalmazhatóságát: mivel nincs szükség egyedi konfigurációra vagy fejlesztésre a forrás oldalon, az architektúra bármilyen kompatibilis MoQ rendszert használó környezetbe azonnal, a meglévő komponensek cseréje nélkül integrálható.
- **MoQ-konform kommunikáció:** A rendszer komponensei közötti vezérlésnek és adatátvitelnek szigorúan a MoQ protokoll alapelveire kell épülnie. Tilos a MoQ mechanizmusain kívüli, „side-channel” (pl. REST API, közvetlen TCP kapcsolat) vezérlőcsatornák alkalmazása. A kommunikációnak a szabványos *Publish/Subscribe* modellben, a SUBSCRIBE, ANNOUNCE és OBJECT üzenetek szemantikájának megfelelően kell zajlania.

3.2. Nem funkcionális követelmények

A felhasználói élmény és a hálózati fenntarthatóság érdekében a rendszernek az alábbi minőségi kritériumoknak kell megfelelnie:

- **Alacsony késleltetés (Low Latency):** Mivel az élő közvetítés egyik legfontosabb mérőszáma a késleltetés, az átkódolás által bevezetett járulékos időkésleltetésnek minimálisnak kell lennie. A rendszernek ki kell használnia a QUIC protokoll stream-multiplexelési és a CMAF formátum fragmentálhatósági előnyeit.
- **Skálázhatóságot figyelembe vevő tervezés:** Bár az elsődleges cél egy működő prototípus megvalósítása, a rendszertervezés során alapvető követelmény a jövőbeli skálázhatóság szem előtt tartása, mivel maga a MoQ architektúrát is ezen az alapelveken fejleszti. Az architektúra kialakításakor kerülni kell azokat a merev tervezési döntéseket, amelyek később gátolnák a rendszer kiterjesztését vagy a nagyobb nézőszám kiszolgálását lehetővé tevő mechanizmusok integrálását.

3.3. A rendszer határai (Scope)

A feladatkiírás és a megvalósíthatóság korlátai alapján a specifikáció a következő területeket explicit módon kizártja a vizsgálatból:

- **Jogosultságkezelés és hitelesítés:** A dolgozat nem foglalkozik a felhasználók azonosításával és jogosultságaival. A fókusz a technológiai megvalósíthatóságon és a protokoll-szintű működésen van.
- **Komplex orkesztráció és terheléselosztás:** Bár a tervezés figyelembe veszi a skálázhatóságot, a dolgozat nem tér ki egy szerver-orkesztrációs rendszer (pl. Kubernetes alapú dinamikus Átkódoló skálázás) implementálására.

3.4. Sematikus rendszerkoncepció

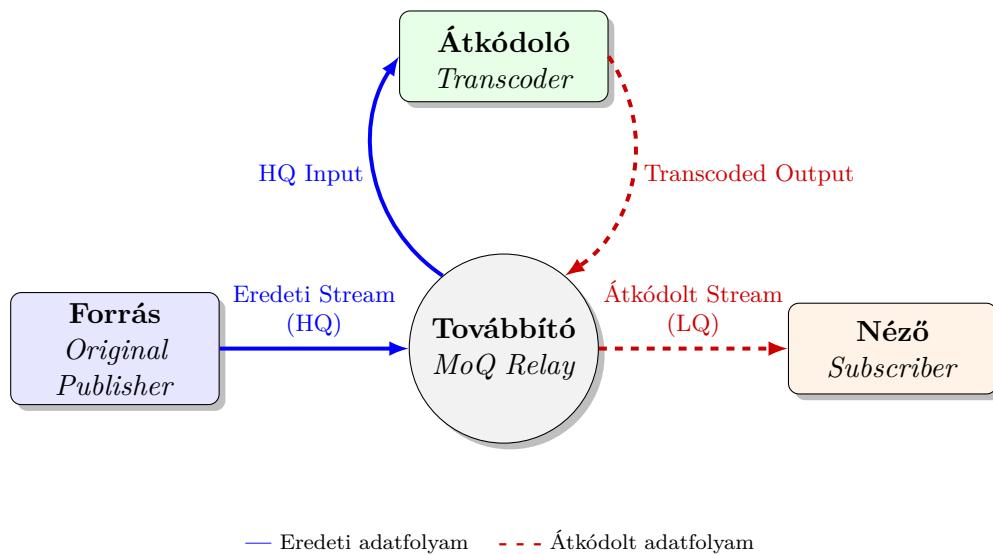
A rendszerterv alapja, hogy a médiaadaptáció terhét levesszük a végfelhasználói eszközökről (Forrás), és azt a hálózatba integrált intelligens csomópontokra helyezzük át. Az alábbiakban vázolom a tervezett architektúra logikai felépítését és a névterek kezelésének elvét.

3.4.1. Architekturális szerepkörök

A koncepció négy fő entitásra bontja a rendszer felépítését. Míg a publikáló és a néző egyszerű végpontként (Client) csatlakozik, a továbbító pedig a hálózati infrastruktúra részét képezi, addig az átkódoló egy speciális hálózati infrastruktúra elem, amely elvi szinten egy végpont, de gyakorlatilag nem gyárt eredeti tartalmat, és nem is végső fogyasztója annak. A szerepkörök a következők:

1. **Eredeti Publikáló (Original Publisher):** A tartalom forrása (pl. egy kamera, OBS szoftver vagy egy emulált forrás). Egyetlen, magas minőségű sávot publikál a hálózatra egy adott névtér alatt (pl. bbb). Nem rendelkezik információval az átkódolásról.
2. **MoQ Továbbító (Relay):** A hálózat közbenő csomópontja. Nem végpont, hanem a végpontok közötti összeköttetést biztosító elem, amely a SUBSCRIBE és OBJECT üzenetek továbbításáért, a kérések aggregálásáért, valamint a gyorsítótárazásáért felel.

3. **Átkódoló (Transcoder):** A rendszer speciális, kettős szerepkörű eleme.
 - Mint **Feliratkozó:** Figyeli a hálózatot, és igény esetén feliratkozik az *Eredeti Publikáló* sávjára.
 - Mint **Publikáló:** A beérkező adatfolyamot dekódolja, megváltoztatja (pl. átméretezi), majd új sávként, egy módosított névtér alatt visszajuttatja a hálózatra.
4. **Néző (Subscriber):** A tartalom fogyasztója. Amennyiben az eredeti forrás paraméterei (pl. felbontás, sávszélesség-igény) nem megfelelőek számára, egy erre a célra specifikált mechanizmus útján, a MoQ protokollon keresztül továbbítja egyedi átkódolási kérését a hálózat felé.



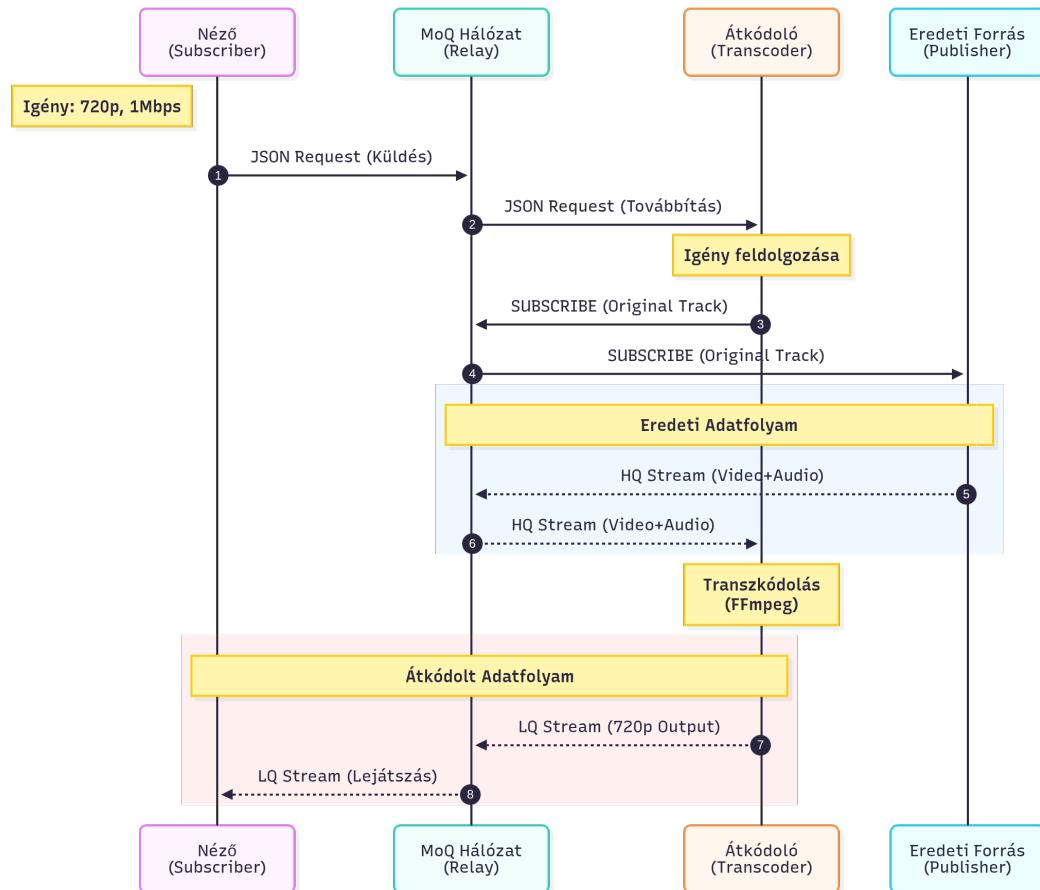
3.1. ábra. Az adatfolyam útja: A Relay továbbítja az eredeti forrást az átkódolónak, majd a visszaérkező eredményt a nézőnek.

3.4.2. Működési logika és Információáramlás

A rendszer működése egy igényvezérelt (on-demand) modellre épül, ahol az átkódolás nem előre definiáltan, hanem a nézői igények alapján indul el. A folyamat magas szintű lépései a következők:

1. **Igény felismerése:** A Néző kliens (Subscriber) úgy dönt, hogy a rendelkezésre álló erőforrások (pl. sávszélesség, hardveres dekódolási képesség) nem elegendők az eredeti stream fogadásához.
2. **Igény továbbítása:** A kliens a passzív feliratkozás helyett egy strukturált kérést (Request) állít össze, amelyben pontosan definiálja a számára szükséges paramétereket (pl. „720p felbontás, 1 Mbps bitráta”). Ezt a kérést a MoQ hálózaton keresztül eljuttatja az Átkódoló komponenshez.
3. **Feldolgozás indítása:** Az Átkódoló fogadja az igényt, és amennyiben az adott variáns még nem létezik, feliratkozik az eredeti forrásra, és elindítja a transzkódolási folyamatot.

4. **Tartalom publikálása:** Az elkészült módosított videófolyamot az Átkódoló publikálja a MoQ hálózatba, egy előre definiált névtér alatt.
5. **Lejátszás:** A Néző a kérése nyomán létrejött (vagy már elérhetővé vált) új sávra a szabványos mechanizmusokkal csatlakozik, és megkezdi a lejátszást.



3.2. ábra. Az átkódolási folyamat alapszintű szekvenciadiagramja.

Ez a sematikus modell biztosítja, hogy a rendszer rugalmasan reagáljon a változó nézői igényekre, amellett hogy a forrásoldali komponensek változatlanok maradnak. A nézők számára lehetőség nyílik egyedi igényeik kielégítésére anélkül, hogy az Eredeti Publikáló vagy a hálózati infrastruktúra működését módosítani kellene, mindezt a MoQ protokoll egységes kommunikációs keretrendszerét hasznosítva.

4. fejezet

Tervezési kérdések és döntések

Az előző fejezetben bemutatott architektúra és a rendszerrel szemben támasztott követelmények – különös tekintettel a skálázhatoságra és a transzparenciára – számos olyan tervezési kérdést vetnek fel, amelyekre a szabvány nem ad közvetlen választ. Ahhoz, hogy a közvetítői hálózatba illesztett átkódoló és a kliensek (Feliratkozók) hatékonyan együttműködjenek, döntenünk kell a komponensek számososságáról, a generált tartalmak láthatóságáról és a metaadatok (katalógus) konzisztenciájának fenntartásáról.

Jelen fejezet ezeket a technológiai dilemmákat járja körül, bemutatva az alternatívákat és indokolva a választott megoldásokat.

4.1. Skálázhatoság: Átkódolók száma egy média névtérben

A specifikációban rögzített skálázhatosági követelmény alapvetően meghatározza, hogyan tekintünk a feldolgozó egységekre. Bár a legegyszerűbb implementáció egyetlen, monolitikus átkódolót feltételezne egy adott közvetítés (névtér) mellett, ez a megközelítés éles környezetben gyorsan szűk keresztmetszetté válhat.

A MoQ objektummodellje [13] lehetővé teszi, hogy egy névtéren (*namespace*) belül tetszőleges számú sáv (*track*) létezzen. Ez lehetőséget ad arra, hogy a különböző minőségű variánsokat (pl. 720p, 1080p) ne egyetlen kliens, hanem több független átkódoló generálja. Ez a megközelítés jelentős előnyökkel járhat, de egyben komplexitást is visz a rendszerbe.

Centralizált (Egy átkódoló) megközelítés Ebben az esetben minden kérés egyetlen központi egységhez fut be. Ez jelentősen egyszerűsíti a rendszerállapot kezelését: nem fordulhat elő névütközés a sávok között, és a naplázás is triviális. Ugyanakkor ez a modell „Single Point of Failure”-t jelent: ha az átkódoló túlterhelődik vagy leáll, az összes alternatív minőség elérhetetlené válik, megsértve ezzel a megbízhatóságra vonatkozó elvárásokat.

Elosztott (Több átkódoló) megközelítés Ha engedélyezzük, hogy egy névtérbe több független átkódoló is publikáljon, a rendszer horizontálisan skálázhatóvá válik. Külön gépek dedikálhatók a nagy számításigényű feladatokra (pl. 4K átkódolás), vagy speciális hardvert igénylő feladatokra és az egyik egység kiesése nem rántja magával a teljes szolgáltatást. Cserébe viszont meg kell oldani a koordinációt: biztosítani kell, hogy két átkódoló ne kezdje el ugyanazt a feladatot párhuzamosan (felesleges erőforrás-használat), és ne ütközzenek a sávnevek.

A döntés Bár a szakdolgozat keretein belül megvalósított prototípusban nem célom skálázást támogató implementáció fejlesztése, az architektúrát a **több átkódolós modell**

4.1. táblázat. Az architekturális döntés összehasonlítása

	Előnyök	Hátrányok
Egy átkódoló	Egyszerű implementáció; Nincs versenyhelyzet és névütközés; Könnyű hibakeresés.	Korlátozott kapacitás (CPU/GPU limit); Egyponos hibalehetőség; Nehézkes bővíthetőség.
Több átkódoló	Horizontális skálázhatóság; Redundancia; Specializálható erőforrások.	Bonyolult koordináció; Katalógus-konzisztenzia fenntartása nehéz; Magasabb üzemeltetési komplexitás.

figyelembevételével terveztem meg. Ez a gyakorlatban azt jelenti, hogy a rendszer nem feltételezi a kizárolagosságot: a névterek és a katalógus kezelését úgy alakítottam ki, hogy az később akadálytalanul támogassa több feldolgozó egység párhuzamos működését.

4.2. Működési hatókör: Egyetlen forrás-névtér elve

A skálázhatóság kérdésköréhez szorosan kapcsolódik az átkódoló komponens belső logikai felépítése. Felmerül a kérdés: egyetlen futó átkódoló példány képes legyen-e párhuzamosan több, egymástól független forrás-névteret (pl. „meeting-A” és „koncert-B”) is kezelni, vagy korlátozzuk a működését egyetlen dedikált forrásra?

A tervezés során az *egyetlen forrás-névtérhez kötött működés* mellett döntöttem, az alábbi indokok alapján:

- Kliens-jellegű működés:** Az átkódoló a MoQ hálózat szempontjából egy kliens (feliratkozó és publikáló egyben). A kliensoldali logika jelentősen egyszerűsödik, ha a belső állapotgépet nem kell megosztani több, potenciálisan eltérő paraméterekkel rendelkező forrás között.
- Orkesztráció és izoláció:** Modern szerverkörnyezetben (pl. Kubernetes) az erőforrások elosztása hatékonyabban kezelhető konténer-szinten, mint alkalmazás-szinten. Ha igény merül fel egy új névtér (pl. „koncert-B”) feldolgozására, logikusabb egy új, tiszta állapotú átkódoló példányt indítani, mint a meglévő, esetlegesen már terhelt példányt bonyolítani kontextusváltásokkal.
- Policy és prioritások:** Ha egyetlen komponens több névteret szolgálna ki, komplex belső ütemezési szabályokat (policy) kellene implementálni arra az esetre, ha az erőforrások fogyni kezdenek (melyik névtér élvezzen előnyt?). A dedikált példányok esetében ez a döntés az orkesztrációs rétegbe kerül, ahol transzparens módon kezelhető.

Ezzel a döntéssel az átkódoló egy könnyűsűlyű, jól definiált feladatot ellátó egység marad, amely a „root” média névterek közötti ugrálás helyett a rábízott tartalom minőségi kiszolgálására fókusztál.

4.3. Erőforrás-hatókonyság: Privát vagy Nyilvános sávok?

A rendszer egyik központi eleme az igényvezérelt működés. Felmerül azonban a kérdés: ha egy felhasználó kér egy speciális formátumot (pl. 480p), az eredmény kizárolag az ő számára legyen elérhető, vagy mások is csatlakozhassanak hozzá?

4.3.1. A „Privát sáv” megközelítés

Kézenfekvőnek tűnhet, hogy minden kérésre egy dedikált adatfolyamot indítsunk (pl. a sáv nevébe kódolva a kérő session-azonosítóját). Ez technikailag egyszerűsítené a jogosultságkezelést és a személyre szabást. Azonban ez a megközelítés szöges ellentében áll a MoQ és a modern videóterjesztés (CDN) alapelveivel. Ha tíz felhasználó kéri ugyanazt a 480p-s videósávot, a privát modellben tízszer végeznénk el ugyanazt a számításigényes átkódolást, és tízszer terhelnénk a hálózatot ugyanazzal az adattal ami súlyos erőforrás-pazarlás.

4.3.2. A „Nyilvános sáv” és a cache-hatékonyság

A választott megoldás a **nyilvános (megosztott) sávok** modellje. Ha egy kliens kezdeményez egy átkódolást, az eredményül kapott sáv bekerül a publikus névtérbe. Ha később egy másik kliensnek ugyanezekre a paraméterekre van szüksége, a rendszer nem indít új folyamatot, hanem kiszolgálja őt a már futó adatfolyamból.

Ez a döntés több előnnyel jár:

- **Számítási kapacitás kímélése:** Egy variánst csak egyszer kell legenerálni.
- **Relay és Cache optimalizáció:** A MoQ Továbbítók hatékonyan tudják továbbítani a csomagokat, így a hálózati terhelés nem nő lineárisan a nézők számával.
- **Dinamikus adaptivitás:** A kliensek – hasonlóan a HLS/DASH működéshez – szabadon válthatnak a már létező sávok között hálózati ingadozás esetén.

Ez a döntés azonban szükségszerűen maga után von egy újabb kihívást: hogyan tudassuk a többi klienssel, hogy egy új, általuk nem kért, de számukra is hasznos sáv jelent meg a rendszerben? Itt lép be a katalóguskezelés kérdése.

4.4. Dinamikus metaadatok: A Katalógus kezelése

Ahogy a 2. fejezetben láttuk, a MoQ kliensek a katalógusból tájékozódnak az elérhető sávokról. A mi esetünkben azonban a katalógus nem statikus: a nézői igények hatására folyamatosan új sávokkal bővülhet.

Mivel döntésünk értelmében több átkódoló is működhet párhuzamosan, és mindegyikük hozhat létre új sávokat, kritikus kérdés, hogyan frissítjük ezt a központi leírót anélkül, hogy inkonzisztens állapotot idéznénk elő.

4.4.1. Lehetséges stratégiák

1. **Közös katalógus sáv közvetlen írása:** minden átkódoló ugyanabba a katalógus sávba próbál írni. Ez elosztott környezetben, központi zár (lock) nélkül szinte garantáltan adatvesztéshez vagy kontinuitási hibákhoz vezet. Mindezek mellett a MoQ specifikáció nem is támogatja ezt a modellt, hiszen egy sáv egyetlen entitás tulajdoná kell, hogy legyen.
2. **Kliens oldali összefésülés:** minden átkódoló saját „mini-katalógust” publikál. A kliensnek kellene feliratkoznia az összesre, és a saját oldalán egyesíteni azokat, azonban ez azt igényelné, hogy a kliens minden átkódoló delta frissítés sávjára feliratkozzon, a teljes delta frissítés történetet lekérje és helyben fésülje össze az adatokat. Ez feliratkozó kliensben egy jelentős változtatást igényelne, amit nem zártunk ki a specifikáció során, de nem lenne logikus döntés hiszen ez egy jelentős mértékű változtatás és esetlegesen sok átkódoló jelenlétében sok kapcsolatot kellene fenntartani

a kliensnek. Mindemellett nehéz lenne garantálni a konzisztenciát és sorrendiséget a frissítések között.

3. **Dedikált Katalógus Készítő komponens:** Egy önálló szolgáltatás, amely összegegyüji az információkat a forrástól és az átkódolóktól, majd egyetlen, hiteles, időrendben konzisztens katalógus folyamot állít elő a kliensek számára.

4.4.2. A választott megoldás: Katalógus Készítő (Catalog maker)

A rendszer integritása érdekében a **harmadik opciót**, a dedikált Katalógus Készítő bevezetését választottam. Ez a komponens működik az „egyetlen igazságforrásként” (single source of truth), aki minden információval rendelkezik ami a média névtérben releváns, hasonlóan ahhoz, mint ahogy a MoQ-ARCH draft-ban is le van írva[12]c.

A működés elve a következő:

- A forrásoldali publikálótól megszerzi a Katalógus Készítő az eredeti katalógus sávot.
- Létrehozza a Katalógus Készítő a saját, dedikált katalógus sávját, amelyre minden végfelhasználó, minden átkódoló kliensek fel tudnak iratkozni.
- Amikor egy átkódoló létrejön, és új sávot publikál, jelzést küld MoQ felett a Katalógus Készítőnek, aki feliratkozik az átkódoló katalógus sávjára. Ezáltal az átkódolók által létrehozott delta frissítéseket fel tudja dolgozni.
- A Katalógus Készítő összefésüli a beérkező delta frissítéseket;
- Konzisztens állapotban továbbítja a frissítéseket a feliratkozó kliensek felé, egy forrásban kezelve minden elérhető sávot.

Ez a megközelítés leveszi a terhet a kliensekről, valamint egyetlen helyen elérhető a teljes katalógus, megkönnyítve ezzel a karbantartást és a bővítést.

4.5. A felhasználói igények jelzése: Implicit vs. Explicit modell

A rendszer egyik legkritikusabb tervezési pontja a kommunikációs interfész meghatározása a végfelhasználóként feliratkozó és az Átkódoló között. Hogyan jelezze a kliens, hogy számára a jelenleg elérhető sávok nem megfelelők, és egy egyedi variánst igényel?

A tervezés során két alapvető megközelítést vizsgáltam: az implicit, névtérbe kódolt igényjelzést, valamint az explicit, strukturált adatobjektumokkal történő vezérlést.

4.5.1. Az implicit, névtér alapú megközelítés korlátai

Kézenfekvőnek tűnhet a MoQ *Subscribe* mechanizmusát használni közvetlen igényjelzésre. Ebben a modellben a kliens egyszerűen megróbál feliratkozni egy olyan sávra, amely még nem létezik, de a neve leírja a kívánt paramétereket. Például: `meeting-123/video/720p/30fps/2mbps`.

Bár ez a megoldás minimalizálná a kliensoldali fejlesztési igényt (hiszen csak a feliratkozási URL-t kellene manipulálni), számos architekturális problémát vet fel:

- **Merev névtér szerkezet:** A paraméterek (felbontás, képkockasebesség, bitráta, kodek) formátumát szigorúan rögzíteni kellene. Egy új paraméter bevezetése (pl. preset: fast) felborítaná a névtér értelmezőket.
- **Korlátozott kifejezőerő:** A névtér karakterlánca nem feltétlen alkalmas komplex konfigurációk átadására.

4.5.2. A választott megoldás: Explicit JSON alapú kérés

A fenti korlátok miatt a rendszerben egy **explicit kéréskezelési modell** mellett döntöttem. A kliens nem a hiányzó sávra iratkozik fel vakon, hanem egy dedikált vezérlő sávra küld egy strukturált üzenetet.

A választott formátum a **JSON**, az alábbi indokok alapján:

1. **Modularitás és Bővíthetőség:** A JSON struktúra lehetővé teszi tetszőleges számú paraméter átadását anélkül, hogy a névtér szerkezetét módosítani kellene. Korábbi meglévő paraméterek mellé bármikor felvehető új, akár teljesen eltérő, sokkal komplexebb átkódolási konfiguráció. A fogadó oldal (átkódoló) a számára ismeretlen mezőket egyszerűen figyelmen kívül hagyhatja, így biztosítva a visszafelé kompatibilitást.
2. **Szerializáció és Validáció:** A szöveges útvonalakkal ellentétben a JSON sémák segítségével szigorúan validálható a beérkező igény még a feldolgozás megkezdése előtt. Ez növeli a rendszer biztonságát és stabilitását, mivel a hibás formátumú kérések azonnal elutasíthatók.
3. **A vezérlés és az adatfolyam szétválasztása:** Ezzel a megoldással logikailag és a névtérben is elkülönül a *vezérlő sík* (Control Plane - a kérés) és az *adat sík* (Data Plane - a létrejövő videósáv). A request névtér alatt zajlik az egyeztetés, míg a generált tartalom a publikus névtérbe kerül. Ez tisztább architektúrát eredményez, és megkönnyíti a hibakeresést.

Bár ez a döntés azt jelenti, hogy a kliens oldalon extra logikát kell implementálni a JSON összeállítására és küldésére, ez az egyszeri befektetés megtérül a rendszer rugalmasságában és a paraméterezés szabadságában.

4.6. A válaszadás aszimmetriája és az explicit elutasítás hiánya

Az igényvezérelt rendszer tervezésekor felmerülő fontos kérdés a hibakezelés és a visszautasítás módja. Mi történik, ha egy kliens olyan kérést küld, amely technikailag nem teljesíthető, vagy egy olyan variánst kér, amely már létezik? Szükséges-e explicit hibaüzenetet küldeni a MoQ hálózaton keresztül?

Az explicit visszautasítás mellőzése mellett döntöttem. A rendszer nem küld „Request Denied” vagy „Error” típusú válaszüzeneteket a kérést indító kliensnek az alábbi okokból fakadóan:

- **Ismételt kérések és meglévő tartalmak kezelése:** Amennyiben a rendszerbe olyan átkódolási igény fut be, amelyet korábban egy másik felhasználó már kért (tehát a folyamat már fut), azt a rendszer nem hibaként, hanem megerősítésként értelmezi. Ilyenkor nincs szükség külön visszajelzésre vagy a kérés elutasítására. A „válasz” maga a tartalom elérhetősége: az igényt bejelentő kliens feladata csupán annyi, hogy figyelje a katalógust, és amint megjelenik (vagy ha már ott van) a kívánt paraméterekkel rendelkező videósáv, csatlakozzon rá.
- **Névütközések elvi kiküszöbölése:** A több átkódolós környezetben a sávnevek véletlenszerű egyezése kritikus hibaforrás lehet. Erre azonban egy okos sávnév-generálási stratégia alkalmazásával hatékonyan fel lehet készülni, így a névütközésből fakadó visszautasítási helyzeteket ki lehet zárni.

- **Protokoll-szintű egyszerűsítés:** A MoQ Pub/Sub természete aszinkron. Egy explicit hibaüzenet visszajuttatása a specifikus kérőhöz (aki csak egy a sok közül) bonyolítaná a kommunikációs modellt, miközben a kliens számára az eredmény (nincs új sáv a katalógusban) ugyanaz.

A visszautasítás tehát implicit módon történik: ha a kérésre záros határidőn belül nem jelenik meg megfelelő bejegyzés a katalógusban, a kliens sikertelennek tekintheti a próbálkozást.

4.7. Tartalmak szinkronizációja és azonosítása

Végezetül technikai, de fontos kérdés a forrás és az átkódolt tartalom közötti kapcsolat. A MoQ „loc” tervezete (pl. LOC draft [15]) szigorú megfeleltetést javasol az összetartozó sávok között (pl. csoportazonosítók (Group ID) és objektum azonosítók (Object ID) szoros összefüggésben vannak eltérő minőségű de aznos forrás tartalommal rendelkező sávok között).

Hagyományos műsorszórásnál, ahol a szerver egyszerre kódolja az összes minőséget, ez könnyen betartható. A mi esetünkben azonban az átkódoló egy később, *tetszőleges időpontban csatlakozó* entitás. Nem várható el tőle, hogy visszamenőleg ismerje a forrás sorszámozását, vagy hogy bit pontosan szinkronban tartsa a saját csoportazonosítóit az eredetivel. Különösen akkor, ha a forrásoldali GOP struktúra változik az átkódolás során. Mindezek mellett van olyan eset is, amikor **egy** sávon belül van több minőség is, amit nem lehet ezzel a megközelítéssel kezelní, hiszen egy sávra csak egy kliens publikálhat.

Ezért a tervezés során a **laza csatolás** elvét követem:

- Az átkódolt sáv független sorszámozással, önálló, önmagában is értelmezhető MoQ sávként jelenik meg, de jelölve van az alternatív reprezentációként a katalógusban.
- A szinkronizációt nem a szállítási réteg sorszámai (Csoport/Objektum azonosító), hanem a média-konténerben (CMAF) utazó időbelyegek (Presentation Timestamp - PTS) biztosítják.

Ez a megoldás egyszerűsíti az implementációt és növeli a rendszer hibatűrését, mivel az átkódoló működése nem függ kritikusan a forrás belső számlálónak pontos követésétől, csupán a szabványos CMAF időzítésre kell ügyelnie.

5. fejezet

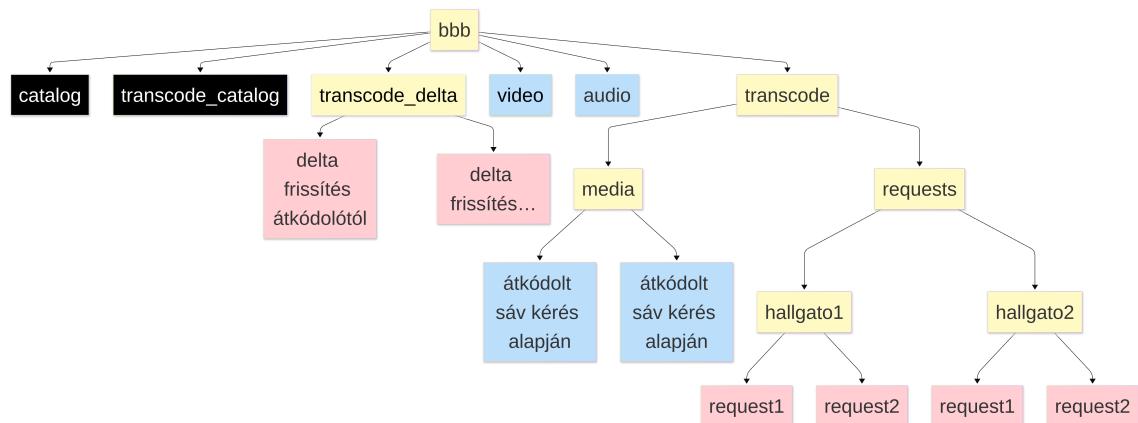
Rendszerterv és protokoll kialakítás

A követelményelemzés és az előző fejezetben tárgyalt tervezési döntések alapján ebben a fejezetben részletezem a megvalósított rendszer architektúráját, a komponensek közötti kommunikációs protokollt, valamint az adatok szervezését biztosító névterek struktúráját. A fejezet célja, hogy pontos technikai leírást adjon arról, hogyan valósul meg az igényvezérelt átkódolás a MoQ hálózaton belül, az adatok címzésétől a vezérlőüzenetek formátumáig.

5.1. Névtér és sávstruktúra tervezése

5.1.1. Kezdeti névtérkoncepció

A rendszer tervezésének korai szakaszában készült kezdetleges koncepciót szemléltet az 5.1. ábra. Ez a struktúra biztosítja a rendszer működéséhez elengedhetetlen adatfolyamok (a metaadatok, a média és a vezérlőjelek) logikai szétválasztását a közös gyökér (a példában bbb) alatt.



5.1. ábra. A névtér-hierarchia és sávstruktúra kezdeti koncepciója

Az ábrán fekete színnel jelölve láthatók a tartalom felfedezhetőségét biztosító katalógus sávok. Az eredeti forrás által publikált szabványos catalog mellett megjelenik a *Katalógus Készítő* komponens által előállított transcode_catalog is. A kliensek számára ez utóbbi biztosítja a transzparenсs elérést, mivel egyetlen felületen egyesíti az eredeti és a rendszer által generált sávok listáját.

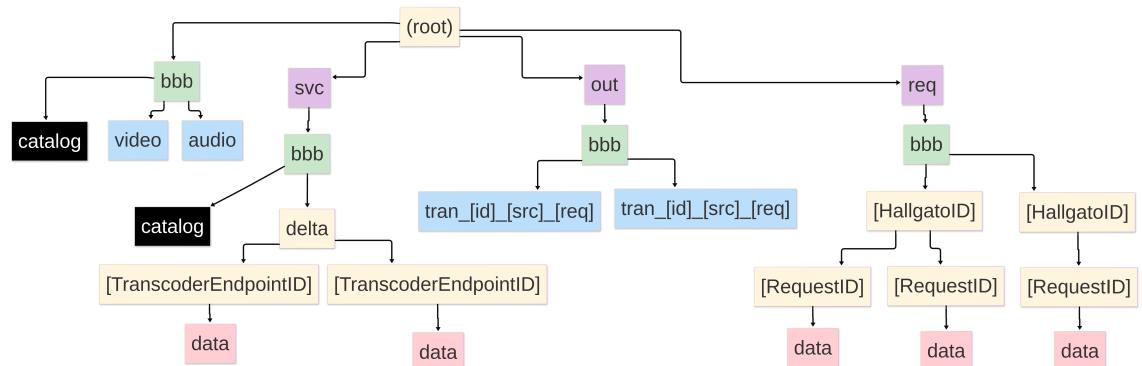
A nagy sávszélességű médiafolyamok kék színnel vannak jelölve. Míg az eredeti videó- és audiotartalmak közvetlenül a gyökér névtérben helyezkednek el (vagy bármely alnévreben), addig az átkódolt variánsok a transcode/media ág alá kerülnek. Ez a szeparálás nem feltétlen szükséges a működéshez, azonban elősegíti a rendszer átláthatóságát és a sávok logikai csoportosítását.

A rendszer belső szinkronizációját a transcode_delta névtér (piros elemek) valósítja meg. Az átkódoló kliensek ide küldik a „delta frissítés” üzeneteket az elkészült sávok technikai paramétereivel, amelyek alapján a Katalógus Készítő folyamatosan frissíteni tudja a központi katalógust.

Végezetül a rendszer dinamikus igénykezelését a transcode/requests ág biztosítja. A hierarchikus felépítés (requests/[hallgatóID]/[requestID]) lehetővé teszi, hogy minden kliens saját, izolált névtérben kommunikáljon. A kérések önálló sávként való megjelenése biztosítja a rendszer terheléselosztását: az igények nem egy dedikált szerverhez kötöttek, hanem bármely szabad kapacitással rendelkező átkódoló feldolgozhatja azokat.

5.1.2. Véleges névtér-kialakítás

A rendszer implementációja során a kezdeti koncepciót a MoQ protokoll specifikációjához és a felhasznált továbbító (Relay) implementációs korlátaihoz kellett igazítani. A véleges, optimalizált struktúrát a 5.2. ábra szemlélteti.



5.2. ábra. A MoQ névterek és sávok véleges elrendezése a rendszerben

A struktúra átalakítását elsősorban a MoQ szabvány hirdetési (Announcement) mechanizmusa indokolta. A protokoll specifikációja szerint ugyanis a hirdetés kizárolag névtér szinten értelmezett, sáv szinten nem. Ez a megkötés alapvetően érintette a rendszer dinamikus elemeit, mint például az egyedi kéréseket vagy az átkódoló végpontok frissítéseit, amelyeket kezdeti hierarchiában egy szinttel feljebb voltak. Mivel sávokat nem lehet meghirdetni, a kliensek nem értesültek volna az új igényekről vagy az új szolgáltatók megjelenéséről. Erre a korlátozásra a hierarchia mélyítése szolgált megoldásként: a korábban sávként elképzelt egyedi azonosítók (például RequestID vagy TranscoderEndpointID) a hierarchiában egy szinttel feljebb kerültek, és önálló névtérre léptek elő. Ezek alatt a névterek alatt a rendszer egy egységesen elnevezett, egyszerű data sávot használ a tényleges információ átvitelére. Ez a megoldás lehetővé teszi a specifikus információkat igénylő kliensek számára, hogy feliratkozzanak egy adott névtér-prefix hirdetéseire, majd a beérkező hirdetés alapján célzottan csatlakozhassanak a szabványosított data sávhöz.

A másik jelentős változtatást gyakorlati okok indokolták. A kezdeti tervekben minden adat (legyen szó kérésről, delta frissítésről vagy az elkészült videóról) logikailag az eredeti tartalom (pl. bbb) alá szerveződött. A tesztelés során azonban kiderült, hogy a továbbító nem képes megbízhatóan kezelni ezt a mélyen tagolt szerkezetet: amikor egyetlen fő ágon

belül keveredtek a különböző mélységen lévő hirdetések, a rendszer működése bizonytalanná vált, és az üzenetek nem mindenkor találtak célba. Bár ez megoldható lett volna a továbbító pontosításával, ez egyszerűt egy komplex feladatot jelentett volna, másrészt pedig ennél egy általánosabb megoldás is alkalmazható volt.

A stabil működés érdekében ezért „kifordítottam” a struktúrát: a funkciók kerültek a hierarchia legtetejére, teljesen elkülönített gyökér-elemekként (lila elemek). A katalógus menedzsment az svc, a felhasználói igények a req, a generált tartalmak pedig az out főkönyvtár alá kerültek. Ebben az új elrendezésben az eredeti tartalom azonosítója (pl. bbb) már nem a fa gyökerét adja (leszámítva az eredeti publikáló elkülönített látóterét), hanem csak második szintű rendező elvként jelenik meg az egyes ágakban. Ezzel a tiszta szétválasztással megszűntek a címzési problémák, mivel ebben a konstrukcióban a hirdetések és feliratkozások azonos hierarchia szinteken történnek, ami által a továbbító kiszámíthatóan tudja kezelni a kommunikációt.

5.1.3. Kimeneti névtér és determinisztikus sávnevek

Az átkódolt tartalmak az out prefix alatt jelennek meg. A rendszer hatékonyságának kulcsa, hogy a kérést indító kliensnek ne kelljen bonyolult üzenetváltásokon keresztül meg tudnia, hogy elkészült-e a videó, és mi annak a neve. Ezt egy determinisztikus névképzési logika biztosítja:

A generált sávok neve a következő formátumot követi:

tran_[TranscoderID]_[SourceTrack]_[RequestID]

Ahol:

- **TranscoderID:** Az átkódoló példány egyedi azonosítója (az esetleges hibakeresés és eredet-megjelölés miatt).
- **SourceTrack:** Az eredeti forrás sáv neve (pl. video), jelezve, hogy miből készült a variáns.
- **RequestID:** A kérést azonosító kulcs, amelyet a kliens generált.

Ez a struktúra (különösen a RequestID szerepeltetése) teszi lehetővé az automatikus felfedezést. Amikor a kliens megkapja a katalógus frissítését, nem kell elemeznie a videó technikai paramétereit. Elegendő egy egyszerű szöveges keresést végeznie a sávnevek között a saját RequestID-jára. Ha talál egyezést, az garantáltan az ő kérése alapján készült sáv, amelyre azonnal feliratkozhat. Ez a mechanizmus minimalizálja a kliensoldali logikát és gyorsítja a lejátszás elindítását.

Entitások azonosítása és hatóköre A névterek hierarchikus felépítése megköveteli a rendszerben résztvevő aktív elemek (kliensek és átkódolók) egyértelmű megkülönböztetését. A tervezés során az azonosítók generálását a felhasználási környezet sajátosságaihoz igazítottam:

- **Kliens azonosítók (HallgatoID):** Mivel a rendszer alapvetően egy streaming architektúrába illeszkedik, feltételezhető egy magasabb szintű felhasználó-kezelő (User Management) réteg megléte. Így a kliensek azonosítására a már meglévő, a rendszerbe bejelentkezett felhasználók egyedi azonosítóit használjuk. Ez biztosítja, hogy a kérések visszakövethetők legyenek egy konkrét nézőhöz.
- **Átkódoló azonosítók (TranscoderID):** Az átkódoló komponensek tervezésekor figyelembe vettetem azt a tényt, hogy optimális esetben ezeket egy szervezett infrastruktúra (pl. felhőszolgáltató) kezeli. Ennek megfelelően az átkódolók egyedi azonosítóit

az üzemeltető környezet könnyen tudja úgy generálni, hogy azok egyediek legyenek a hálózaton belül.

Fontos tényező a döntésben, hogy ezen azonosítóknak nem szükséges globálisan egyedinek lenniük a teljes MoQ hálózaton belül. Elegendő, ha az éppen kiszolgált médiataralom (pl. bbb névtér) kontextusában egyediek. Ez jelentősen egyszerűsíti az azonosítók generálását és az ütközések elkerülését, mivel az érvényességi körük a tartalomra korlátozódik.

5.2. A kérés adatmodelljének gyakorlati terve

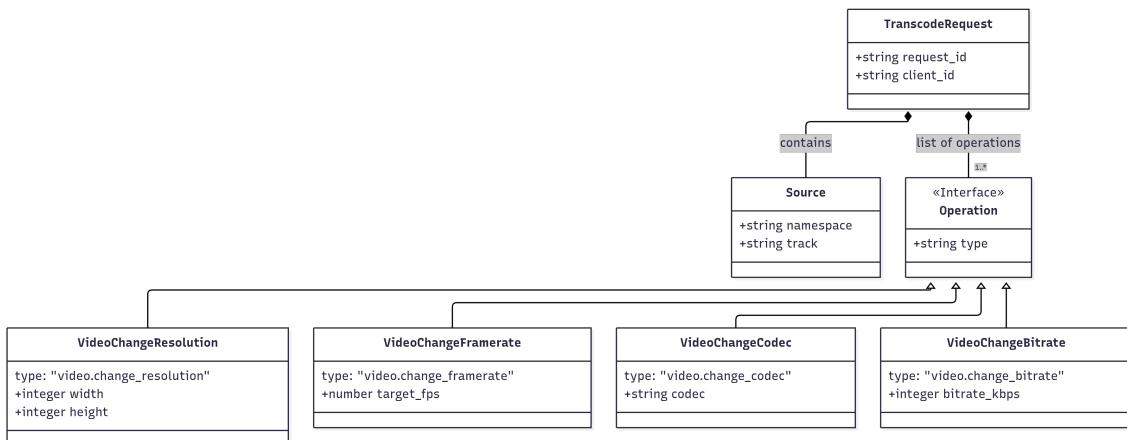
A következő lépés a kommunikációs interfész pontos definíciója. Korábban már el lett döntve, hogy egy explicit objektumba szeretnénk foglalni az átkódolási kérést a paramétereivel. Az adatmodell kialakításában a bővíthetőség játszotta a legfőbb szerepet, hiszen a felbontás változtatás csak egy töredék része annak, amire használni lehetne a rendszert.

5.2.1. JSON formátum választása

Az adatmodell megvalósításához a JSON (JavaScript Object Notation) formátumot választottam, első sorban azért, mert a katalógus formátuma is JSON, így praktikusan nem kell egy architektúrán belül több hasonló de mégis eltérő szintaktikával rendelkező adatstruktúrát kezelni. Ezen felül a JSON széles körben támogatott, könnyen olvasható és írható, valamint számos programozási nyelvben könnyen kezelhető széles körben használt könyvtárakkal.

5.2.2. Adatmodell felépítése

A kérés adatmodelljének központi eleme egy operátor típus, amely meghatározza az átkódolási művelet jellegét. Ez adja a teljes kérés adatstruktúra modularitását, hiszen új operátorok hozzáadásával a rendszer könnyen bővíthető további funkcionálisokkal anélkül, hogy a meglévő struktúra logikáját módosítani kellene. A jelenlegi operátorok paramétereinek deklaratív, tehát ténylegesen nem műveleteket definiálnak, hanem a kívánt állapotot írják le. Ettől függetlenül semmi se gátolja meg a jövőben hogy olyan operátort is bevezessünk, amely nem deklaratív módon működik, hanem explicit végrehajtandó utasításokat tartalmaz.



5.3. ábra. A kérés JSON adatmodell diagramja

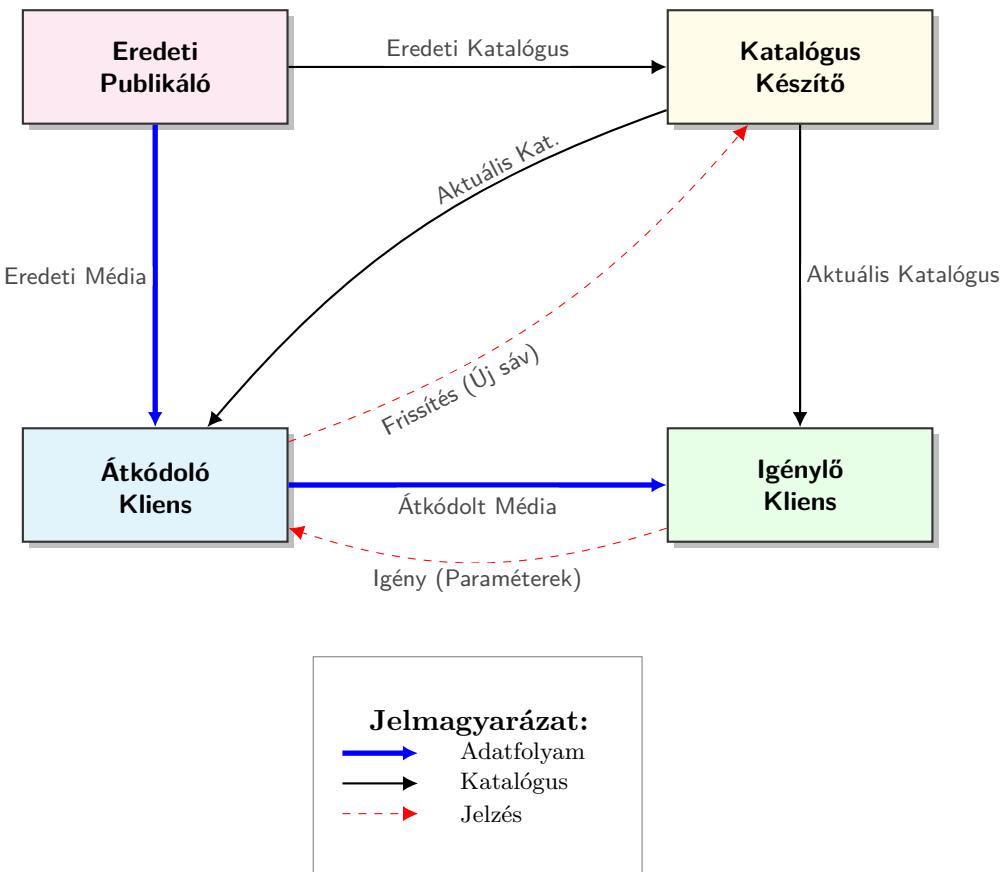
Ezen felül szükség lehet arra is, hogy meghatározzuk a forrást, hiszen ha több különböző videóforrás van egy média közvetítésben (pl. más kamera állások, egy meeting esetén más résztvevők), akkor egyértelműsíteni kell, hogy melyik forrást szeretnénk átkódolva. Ezt a célt szolgálja a source_id mező, amely egy forrás sáv névterét nevét várja paraméterek.

A teljesség és visszakövethetőség érdekében a kérés modell tartalmaz egy egyedi azonosítót (request_id), illetve a kérést létrehozó kliens azonosítóját (client_id) is, amelyek megegyeznek a névtér struktúrában használt értékekkel.

A 5.3. ábra szemlélteti a kérés adatmodelljének felépítését. A megjelenítés egyszerűségének értekében nem tartalmazza az összes operátor amit példának készült, de a lényeget bemutatja. A teljes JSON séma az F.2-es mellékletben található.

5.3. Rendszerkomponensek és a kommunikációs folyamat köztük

A rendszer statikus része után ebben a szakaszban a dinamikus működést ismertetem. A rendszer architektúráját és a komponensek közötti üzenetváltások magas szintű áttekintését a 5.4. ábra szemlélteti.



5.4. ábra. A média streaming rendszer architektúrája (Rács elrendezés).

5.3.1. Szerepkörök az architektúrában

A hálózatban négy eltérő alkalmazáslogikát megvalósító kliens entitás vesz részt, valamint egy vagy több továbbító:

- **Eredeti Publikáló (Original Publisher):** A tartalom forrása. Passzív szereplő az átkódolási folyamatban, kizárólag a saját, jó minőségű médiafolyamát és a saját katalógusát publikálja. Nem tud a rendszerben működő átkódolókról.
- **Katalógus Készítő (Catalog Maker):** A rendszer adminisztrációs központja. Feladata az eredeti és a generált tartalmak egységesítése egyetlen, transzparens katalógusba, biztosítva, hogy a kliensek mindenkorának megfelelően használhatják.
- **Átkódoló (Transcoder):** A dolgozó komponens. Figyeli a felhasználói igényeket, elvégzi a számításigényes műveleteket, és publikálja az eredményt, valamint értesíti a Katalógus Készítőt a változásokról.
- **Igénylő Kliens (Request Client / Viewer):** A végfelhasználó. Alapesetben passzív tartalomfogyasztó, de igény esetén aktív publikálóvá válik, amikor beküldi az átkódolási kérést.
- **Továbbító (Relay):** A hálózati infrastruktúra eleme. Nem végez alkalmazásszintű logikát, feladata kizárólag a névterek hirdetéseinek (ANNOUNCE) és az adatfolyamok (SUBSCRIBE, OBJECT) továbbítása a megfelelő végpontok felé.

5.3.2. Részletes kommunikációs folyamat

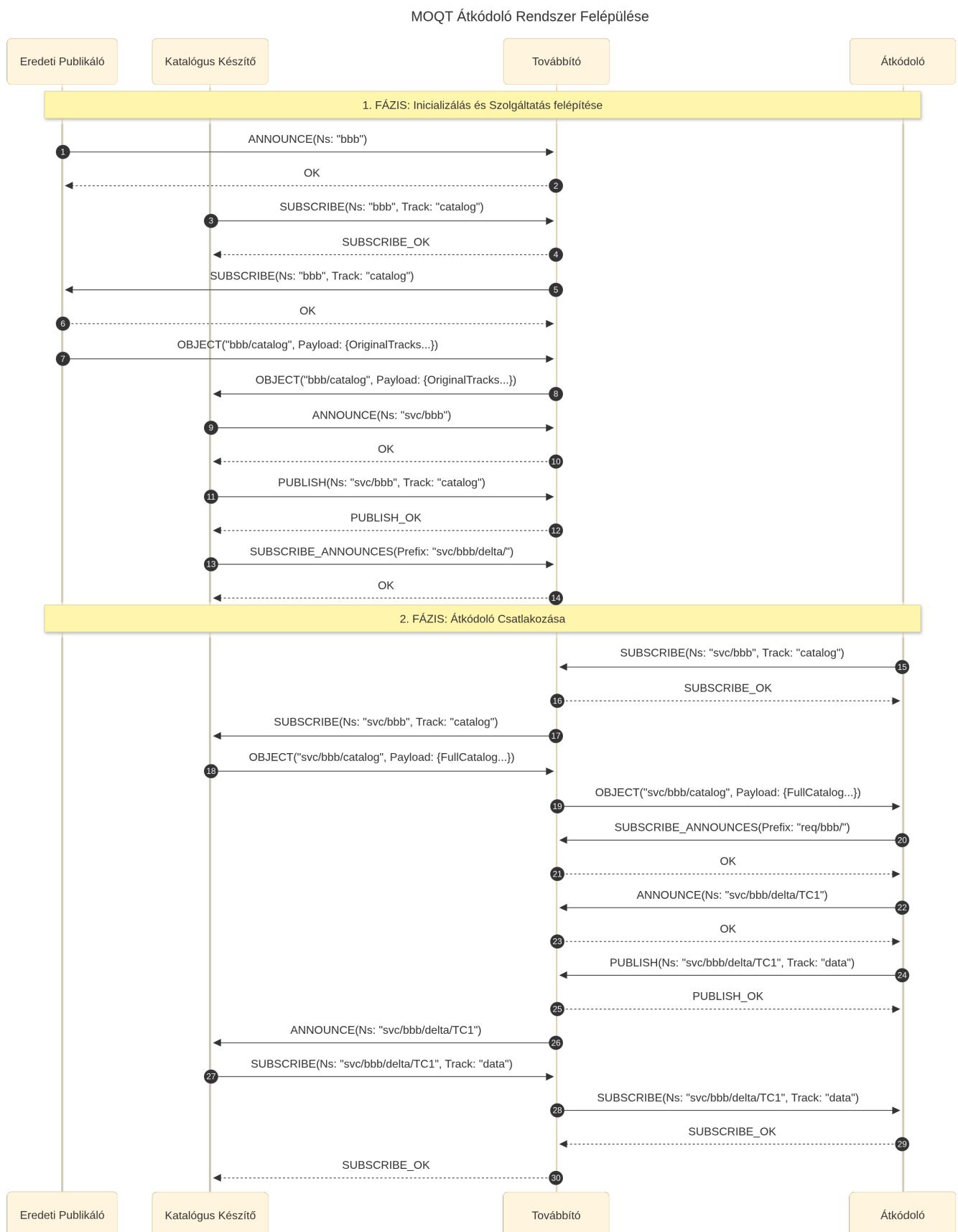
A rendszer működése egy eseményvezérelt láncolat, amely jól elkülöníthető fázisokra bontható. Az alábbiakban végigkövetjük egy átkódolási igény életciklusát az inicializálástól a lejátszásig.

1. Fázis: Inicializálás és a Szolgáltatás felépítése A rendszer működésének alapjait az 1. fázis teremti meg (lásd 5.5. ábra felső szakasza). A folyamat az **Eredeti Publikáló** indulásával kezdődik, amely a hálózaton elérhetővé teszi a bbb névteret. Ezt követően csatlakozik a **Katalógus Készítő**, amely a diagram 3-8. lépéseinben kiépíti a szolgáltatás gerincét:

- Feliratkozik az eredeti publikáló katalógusára, hogy megismerje a forrásanyagot.
- Publikálja a saját, „bővített” katalógus sávját az svc/bbb névtérben (9-12. lépés), amelyhez a későbbiekben a nézők csatlakozni fognak.
- Feliratkozik az svc/bbb/delta névtér-prefix hirdetéseire (13-14. lépés). Ez a lépés biztosítja, hogy a Katalógus Készítő értesüljön minden, a jövőben csatlakozó átkódoló jelenlétéiről.

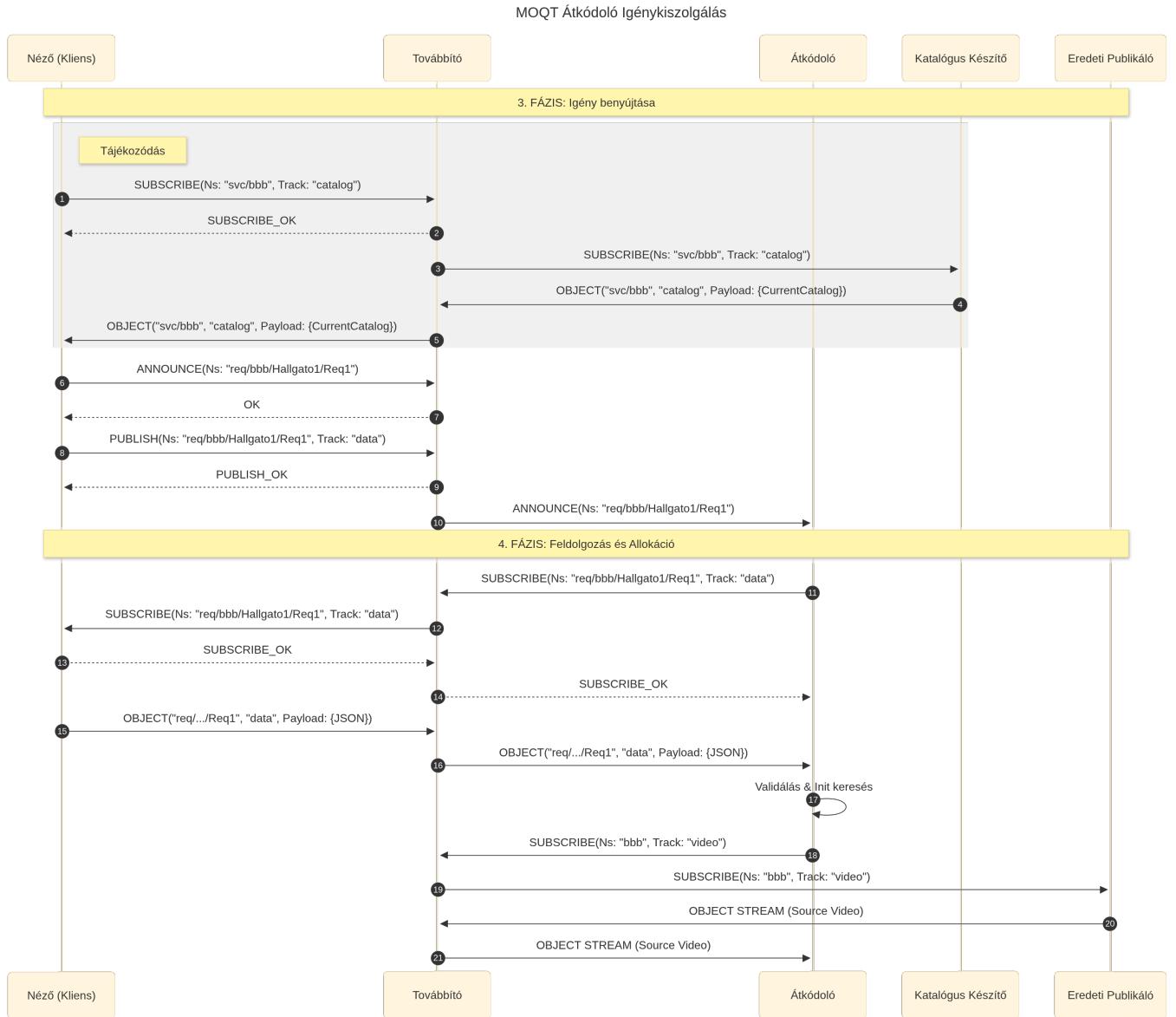
2. Fázis: Az Átkódoló csatlakozása és regisztrációja A 5.5. ábra alsó szakasza mutatja be, hogyan integrálódik a rendszerbe egy új **Átkódoló** kliens. Indulásakor először tájékozódik a hálózat állapotáról:

- A 15-19. lépésekben feliratkozik a **Katalógus Készítő** által publikált bővített katalógusra, így minden naprakész információval rendelkezik a forrásokról.
- Feliratkozik a req/bbb névtér hirdetéseire (20-21. lépés). Ezzel biztosítja, hogy azonnal értesítést kapjon, amint bármelyik felhasználó igényt nyújt be.



5.5. ábra. Rendszer inicializálása és a jelzési útvonalak kiépülése
(1. és 2. Fázis)

- Létrehozza a saját kommunikációs csatornáját az svc/bbb/delta/[TranscoderID] névtér alatt, és meghirdeti azt (22-26. lépés).
- A hirdetés (ANNOUNCE) hatására a Katalógus Készítő értesül az új átkódolóról, és a 27-30. lépésekben feliratkozik annak dedikált data sávjára.



5.6. ábra. Átkódolási igény benyújtása és feldolgozása (3. és 4. fázis)

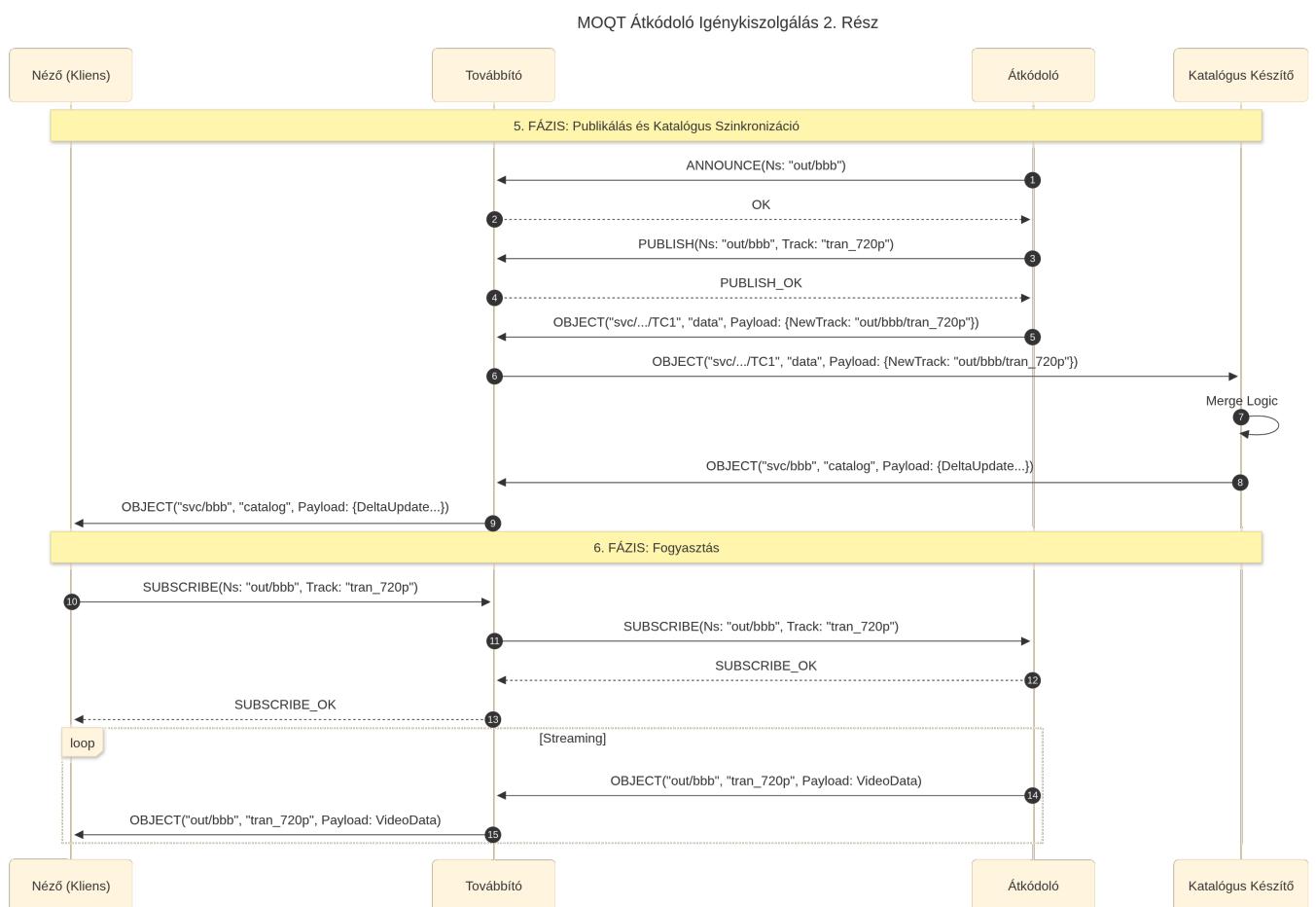
3. Fázis: Az igény benyújtása A folyamat a Néző (Request Client) belépésével folytatódik, ahogy azt a 5.6. ábra felső része mutatja. Mielőtt bármit kérne, tájékozónia kell az elérhető tartalmakról:

- **Katalógus letöltése:** A diagram 1-5. lépéseiben a Néző feliratkozik a **Katalógus Készítő** által publikált bővített katalógusra. Ez a lépés két szempontból fontos, mivel ezáltal nemcsak a forrásokat ismeri meg, de ezen a csatornán keresztül értesül majd a kérése teljesüléséről is.

- Igény felismerése:** A letöltött katalógusban a Néző látja az eredeti forrást, de nem találja a számára szükséges formátumot (pl. 720p).
- Kérés generálása:** A katalógusból kinyert forrás-azonosító (Source Track ID) alapján összeállítja a TranscodeRequest JSON objektumot.
- Kérés publikálása:** A 6-9. lépésekben létrehoz és meghirdet egy egyedi névteret a req/bbb/[HallgatoID]/[RequestID] hierarchiában, majd a benne lévő data sávra publikálja a JSON kérést.

4. Fázis: Feldolgozás és Allokáció Mivel az Átkódoló a 2. fázisban feliratkozott a req névtér hirdetéseire, a Továbbító a 10. lépésben azonnal eljuttatja hozzá a kliens hirdetését.

- Az Átkódoló a 11-16. lépésekben feliratkozik a kliens által meghirdetett sávra, és letölti a kérést.
- A JSON tartalom validálása után (17. lépés) a saját katalógus-példánya alapján azonosítja a kért forrást, és megszerzi a bemeneti média technikai paramétereit.
- Végül a 18-21. lépésekben feliratkozik a forrás videósávjára, és elindítja a belső FFmpeg szálat a kért paraméterekkel.



5.7. ábra. Publikálás és fogyasztás (5. és 6. fázis)

5. Fázis: Publikálás és Katalógus-szinkronizáció Amint az átkódolás elindul és az első médiagyökerek elkészülnek, a rendszer a 5.7. ábrán látható módon reagál:

- Az Átkódoló az 1-4. lépésekben elkezdi publikálni az új videófolyamot az out névtér alatt.
- Ezzel párhuzamosan (5-6. lépés) összeállít egy **Delta Update** üzenetet az új sáv adataival, amelyet elküld a saját delta sávján.
- A Katalógus Készítő – amely hallgatja ezt a sávot – fogadja az üzenetet, a 7. lépésben beilleszti az új bejegyzést a központi adatbázisába.
- **Értesítés:** A 8-9. lépésben a Katalógus Készítő kiküldi a frissítést a bővített katalogus sávon. Mivel a Néző a 3. fázisban már feliratkozott erre a sávra, a Továbbító azonnal továbbítja neki ezt a frissítést.

6. Fázis: Fogyasztás A folyamat lezárásaként a kliens megkapja a frissítést (9. lépés). A katalógusban megjelenő új bejegyzés alapján azonosítja, hogy az általa kért tartalom elkészült. A 10-12. lépésekben feliratkozik az új, out névtérben lévő sávra, és a 14-15. lépéstől kezdve megkezdődik a folyamatos lejátszás.

6. fejezet

Implementáció

6.1. Fejlesztési környezet

A rendszer implementációja elsődlegesen C++ nyelven készült, amelynek hálózati kommunikációs rétegét a MoQ szabványtervezetét megvalósító **LibQuicR** könyvtár biztosítja. A médiaadatok kezelése külső ipari szabványokra támaszkodik: az átkódoló a beérkező CMAF szegmensek manipulációját (dekódolás, skálázás, kódolás) az FFmpeg keretrendszerrel végzi, míg a kliensoldali videólejátszássért és a hálózati stream folyamatos megjelenítéséért a GStreamer felel. A komponensek közötti strukturált adatcsere (katalógus, kérések) szerializációs rétegét az nlohmann/json könyvtár valósítja meg[9].

6.2. Kiindulási állapot

A fejlesztés alapját egy már létező, működőképes MoQ ökoszisztéma képezte. A kiindulási architektúra részét képezte egy publikáló komponens, amely képes volt egy bemeneti, multiplexált streamet önálló sávokra bontani, ezekről szabványos katalógust generálni, majd a tartalmat a hálózaton közzétenni. Ezzel párhuzamosan rendelkezésre állt egy feliratkozó kliens is, amely a fogadott katalógus feldolgozása után képes volt a kiválasztott hang- és videósávokra feliratkozni, valamint azokat a felhasználó számára megjeleníteni. Ebből a statikus modellből azonban hiányzott a dinamizmus, mivel a rendszer kizárolag a forrás által eredetileg előállított minőséget tudta továbbítani a kliensek felé. Az implementáció keretében tehát az ezeken felül korábbiakban definiált komponensekkel bővült ki a rendszer.

6.3. A Catalog Maker megvalósítása

A Catalog Maker a rendszer adminisztrációs csomópontja, amelynek elsődleges feladata a heterogén forrásokból – az eredeti publikálótól és a dinamikusan belépő átkódolóktól – származó információk egységesítése. Implementációs szempontból ez a komponens egy hibrid MoQ végpontként viselkedik: egyszerre lát el *Subscriber* (adatgyűjtés a forrásoktól) és *Publisher* (az egységesített katalógus terítele a nézők felé) feladatokat.

6.3.1. Belső architektúra és adatmodell

A komponens központi eleme a CatalogManager osztály, amely egy memóriában tárolt, egységesített katalógus-modellt (`unified_catalog_`) kezel. Ez az objektum reprezentálja a rendszer aktuális, globális állapotát. A megvalósítás három párhuzamos adatcsatorna kezelésére épül: a forrásoldali katalógus változásainak követése, az átkódolók által küldött

JSON Patch formátumú részleges frissítések fogadása, valamint a konszolidált katalógus publikálása a kliensek felé. Mivel a beérkező események (új sáv létrejötte, forrás frissülése) aszinkron módon, a hálózatról érkeznek, az adatok konzisztenciáját a CatalogManager egy mutex-szel védett kritikus szakaszban végzett módosításokkal garantálja.

6.3.2. Eseményvezérelt működési logika

A Catalog Maker belső logikája a MoQ hálózatról érkező jelzésekre reagáló eseményvezérelt modellre épül. A rendszer működése három fő fázisra bontható.

Az **inicializálás** során a komponens meghirdeti saját szolgáltatási névterét (`svc/<root>/catalog`), majd feliratkozik két kritikus forrásra: az eredeti publikáló katalógusára, valamint a `svc/<root>/delta` névtér-prefixre (ebben a kontextusban a `<root>` megegyezik a korábbi „bbb” névtérrel). Ez utóbbi teszi lehetővé a hirdetés figyelést, amellyel a rendszer a későbbiekben csatlakozó átkódolókat detektálja egy callback függvényben. Ezt követi a **szinkronizációs fázis**, amelyben a rendszer az eredeti forráskatalógus megérkezésére vár. Miután megérkezett átvált az aktív fázisba.

Az **aktív fázisban** a rendszer üzemszerűen működik, és három típusú eseményt kezel. Amikor a Relay jelzi, hogy új névtér jelent meg a figyelt delta prefix alatt, a rendszer dinamikusan létrehoz egy új kezelőt (`DeltaInputHandler`), és feliratkozik az adott átkódoló adatcsatornájára. Az átkódolótól érkező JSON Patch üzeneteket a rendszer validálja, majd a `ProcessTranscoderDelta` metóduson keresztül alkalmazza a memóriában lévő katalóguson.

6.3.3. Dinamikus selfedezés implementációja

A rendszer skálázhatóságának kulcsa, hogy előre nem ismert számú átkódolót képes kezelni manuális konfiguráció nélkül. A `PublishNamespaceReceived` callback függvény felelős azért, hogy amikor a hálózaton megjelenik egy új átkódoló névtere, a rendszer automatikusan dedikált kezelőt rendeljen hozzá, integrálva az új átkódolót a Katalógus Készítő logikába.

6.4. A Transcode kliens belső logikája

Az átkódoló végzi a teljes architektúrában a számításigényes munkát. A gyakorlati megvalósítás során a legfontosabb szempont az volt, hogy a hálózati kommunikáció (amelynek gyorsnak és reszponzívnak kell lennie) ne akadjon meg a videofeldolgozás (amely lassabb és erőforrás-igényesebb) miatt. Ezt a problémát egy többszálú, eseményvezérelt architektúrával hidaltam át.

6.4.1. Aszinkron feladatkezelés

Az átkódolás nem egyetlen monolitikus folyamatként működik, hanem egy „dispatcher” (feladatosztó) modellt valósít meg. A főszál feladata kizárolag a hálózat figyelése: folyamatosan hallgatja a MoQ hálózatot, kifejezetten a felhasználói kéréseket figyelve.

A gyakorlatban ez úgy néz ki, hogy az alkalmazás induláskor feliratkozik egy speciális névtér-prefixre. Amikor a hálózat jelzi, hogy valaki, valahol beküldött egy kérést ebben a névtérben, az Átkódoló azonnal reagál:

1. Feliratkozik a sávra és értelmezi a kérést tartalmazó adatcsomagot.
2. Létrehoz egy teljesen izolált, új munkaszálat (worker thread).

- Ez a munkaszál megkapja a feladatot (pl. „készíts 720p-s streamet ebből a forrásból”), és innentől kezdve önállóan dolgozik.

Ez a megközelítés biztosítja, hogy ha egy videó átkódolása megterheli a processzort, az nem lassítja le az új kérések fogadását vagy a többi videó továbbítását. Fontos megjegyezni hogy bár külön szálakon indul el minden átkódolási feladat, megtörténhet, hogy egy időben több szál is ugyanazt a forrást használja, például ha két különböző felhasználó ugyanazt a forrást egymástól eltérő felbontásban kérte. Ebből az okból kifolyólag egy köztes elem lett beiktatva, amely meglévő forrás esetén azt újrahasznosítja, új forrás esetén pedig létrehozza azt.

6.4.2. Média processzálás és adatútvonal

A videó átalakítása az FFmpeg motorra épül, de a legkritikusabb rész az adatok eljuttatása a hálózatról a kódolóba és vissza. Mivel a MoQ hálózaton az adatok csomagokban (objektumokban) érkeznek, az FFmpeg viszont folyamatos bájt-adatfolyamot vár, egy közvetítő puffer került alkalmazásra.

A megvalósított adatútvonal a következő lépésekkel áll:

- Bemeneti Körkörös Puffer:** A hálózatról beérkező videó-objektumokat a rendszer egy memóriában lévő körkörös pufferbe írja. Ez azért fontos, mert kieggyenlíti a hálózati ingadozásokat: ha hirtelen sok csomag jön, a puffer tárolja őket, amíg a kódoló utoléri magát; ha pedig a hálózat lassul, a kódoló a pufferból még tud dolgozni egy ideig.
- Átkódolás:** Az FFmpeg közvetlenül ebből a memóriapufferből olvassa az adatokat. A rendszer dekódolja a képet, elvégzi a kért módosítást (jelenleg csak az átméretezésre van implementált átkódoló logika demonstrációs célból), majd újra tömöríti a H.264 szabvány szerint.
- Kimeneti Fragmentálás:** A kódoló a kimenetén már alapvetően CMAF szegmenket állít elő. Ezeket egy szálbiztos deque-be helyezi. Innen egy külön erre a célra létrehozott szál veszi ki a szegmenseket, és csomagolja őket MoQ objektumokba majd küldi tovább a hálózaton.
- Szinkronizáció és Indítás:** Fontos, hogy mielőtt az első tényleges átkódolt szegmenst elküldenénk, az FFmpeg komponens generál egy inicializáló szegmenst az új formátumról. Ezt Base64 formátumba kódolva elküldi a Catalog Makernek. Ez garantálja, hogy mire a nézőhöz odaér az első videóadat, a lejátszója már tudni fogja, hogyan kell azt dekódolni, hiszen a Catalog Maker delta frissítésében szerez tudomást az elkészült sávról.

6.5. A kéréskezelő logika és a kliens működése

Az implementációban mind az átkódolást igénylő mind pedig az átkódolási igényt fogadó komponens egy közös „request” adatmodellt használ, amelyet a 5.2 részben ismertettem. A komponens pedig elvégzi a háttérünkét, a szerializációt és deserializációt, valamint az adatok érvényességét. Ez utóbbi azért is hasznos, mert így már a küldés pillanata előtt kiderül, ha valaki esetleg negatív felbontást kérne.

Az Igénylő Kliens működése a felhasználó szemszögéből nézve egy logikus folyamatra épül. Amikor a program elindul, első lépésként letölți az aktuális katalógust, és megjeleníti a választható videókat. Videó választás után meg lehet adni a választott felbontást, a kliens a háttérben összeállítja a request objektumot a fent említett közös komponens segítségével.

Ezt a kérést a kliens a korábban definiált sávra küldi ki a hálózatba. Miután a kérés elment, a kliens várakozó módba kerül: nem vár azonnali választ az átkódolótól, hanem a katalógust figyeli. Mivel a rendszer minden új videóról értesítést küld, a kliens azonNAL észreveszi, ha megjelenik egy olyan új sáv, ami illeszkedik az ő kéréséhez. Amint ez megtörténik, automatikusan csatlakozik az új sávra, és elindítja a lejátszást a GStreamer motorral, így a felhasználó ebből az egész háttérfolyamatból csak annyit érzékel, hogy egy rövid várakozás után elindul a kért felbontású videó.

7. fejezet

Kiértékelés

Jelen fejezet célja a megvalósított rendszer gyakorlati validálása. A bemutatott mérések igazolják a rendszer működőképességét, valamint számszerűsítik annak teljesítményét és erőforrás-igényét valós körülmények között.

7.1. Tesztkörnyezet és konfiguráció

A mérések során a fő szempont a rendszer válaszidejének és késleltetésének vizsgálata volt, nem pedig a hálózati környezet hatásainak elemzése. Ennek megfelelően a komponensek minden localhoston kommunikáltak. minden komponenst (Publikáló, Továbbító, Átkódoló, Katalógus Készítő, Igénylő Kliens) külön processzként futtattam az alábbi hardverkonfiguráción:

- **Processzor:** AMD Ryzen 9 5900HX (8 mag / 16 szál, 3.3 GHz)
- **Memória:** 32 GB DDR4
- **Operációs rendszer:** Fedora Linux 43
- **Szoftver:** FFmpeg 7.1.2

Tesztanyagként a *Big Buck Bunny* videót használtam 4K felbontásban, 30 fps-el, 60-as GOP mérettel, kb 16,5 Mb/sec-es bitrátával, amelybe előzetesen egy képkocka-számlálót égettem a késleltetés vizuális méréséhez.

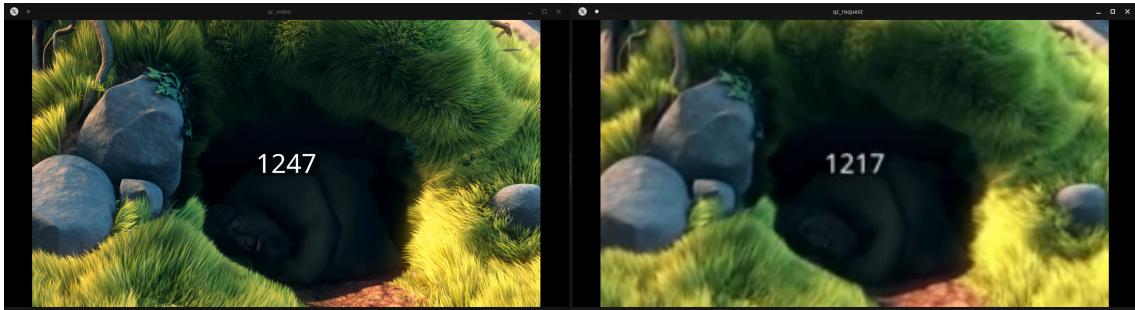
7.2. Funkcionális ellenőrzés

A rendszer alapvető működése sikeres, azaz az igényvezérelt átkódolási lánc önállóan lefut, leszámítva az igényelt felbontás megadását, amit a program a felhasználótól vár el. A folyamat logfájlokkal és vizuális ellenőrzéssel követhető: a kliens által elküldött kérést az átkódoló észlelte, elindította a feldolgozást, majd a katalógus frissítése után a kliens automatikusan átváltott az új sávra.

A 7.1. ábrán látható a rendszer működés közben: a bal oldalon az eredeti 4K forrás, jobb oldalon pedig a kliens kérésére elindult 144p stream fut. A képminőségbeli különbség és a felbontásváltozás egyértelműen jelzi a sikeres transzkódolást.

7.3. Teljesítménymérések

A mérések során a rendszer reakcióidejét és erőforrás-hatékonyságát vizsgáltam.



7.1. ábra. A rendszer működés közben: Bal oldalon az eredeti 4K forrás, jobb oldalon az igényvezérelt 144p átkódolt stream.

7.3.1. Késleltetés (Latency)

A késleltetést két szempontból mértem:

Indítási idő (Request-to-First-Frame) A kérés elküldése és az első képkocka megérkezése között 5 teszt eredménye alapján 436 ms, 290 ms, 182 ms, 167 ms és 298 ms telt el (változóan 720p és 1080p felbontású átkódolási kérésekkel). Ez az időtartam magában foglalja a hálózati kommunikációt, a kérés feldolgozását, az FFmpeg inicializálását, eredeti sávra való feliratkozást, arra az adatok megérkezését és a katalógus frissítését.

Többletkésleltetés (End-to-End) Az eredeti és az átkódolt stream közötti időkülönbséget a képkocka-számlálók összehasonlításával mértem. Az átkódolt stream átlagosan 1 másodperc késéssel követte az eredetit az átkódolt videó indulásakor, ezt a 7.1 ábrán látható 30 képkocka különbség mutatja.

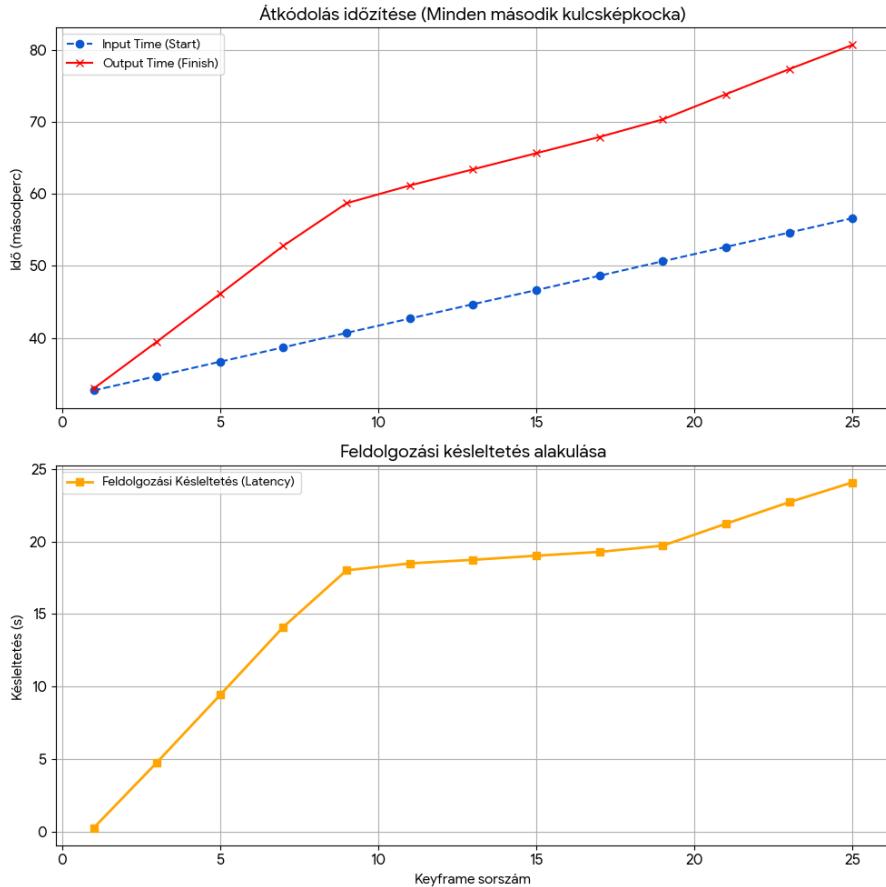
7.3.2. Erőforrás-igény és Pufferkezelés

A processzorterhelés mérése során a 7.1-es táblázaton látható, hogy a rendszer üresjáratban (Idle) elhanyagolható erőforrást igényel, a terhelés pedig a párhuzamos átkódolások számával arányosan növekszik.

Állapot	CPU Terhelés (%)
Idle (Hálózat figyelése)	< 1%
1 aktív stream (4K → 720p)	≈ 9,5%
2 aktív stream (4K → 720p és 4K → 1080p)	≈ 21%

7.1. táblázat. A Transcode kliens CPU terhelésének alakulása

A teljesítményelemzés során logoltam a beérkező adatok pufferbe kerülésének idejét és a kimeneti fragmentek elkészültét, ez a 7.2. ábrán látható. Érdemes megfigyelni rajta, hogy kb. a 9. másodperctől kezdve az Eredeti Publikáló le lett állítva, így a bemeneti adatok érkezése megszakadt, és a feldolgozási idő arányos lett nagyjából a videó sebességével (kb. azonos a meredekség az adatok érkezésével). Ez arra utal, hogy a jelenlegi pufferkezelés nem optimális: amíg az átkódoló aktívan kapja a bemeneti adatokat, a feldolgozás lassabb, mivel nem fér hozzá a pufferhez szabadon az átkódoló motor, míg az adatfolyam leállása után a maradékot nagyjából lejátszási sebességgel dolgozza fel.



7.2. ábra. Pufferkezelés vizsgálata: A bemeneti adatok érkezési ideje (kék pontok) és a kimeneti fragmentek elkészülési ideje (narancssárga pontok).

7.4. Tesztek összegzése

A tesztek eredményei és a fejlesztési tapasztalatok alapján a rendszer teljesíti a specifikációban rögzített követelményeket:

- **Igényvezérelt működés:** A mérések (lásd CPU terhelés) igazolták, hogy a rendszer erőforrás-takarékosan működik: az átkódolási folyamat és a vele járó terhelés kizárálag valós nézői igény esetén jelentkezik.
- **Transzparencia:** Bár ezen követelmény közvetlen tesztelése nehézkes, a koncepció sikerességét bizonyítja, hogy a fejlesztés során az **Eredeti Publikáló** kliens lényegi működését nem volt szükséges megváltoztatni. A forráskódban eszközölt módosítások a MoQ protokoll használatot nem érintették, csupán optimalizációs jellegűek voltak.
- **MoQ kompatibilitás és alacsony késleltetés:** A kommunikáció kizárálag szabványos MoQ üzenetekkel valósult meg, a mért hozzáadott késleltetés pedig legfeljebb olyan nagyságrendekben mozog mint hasonló technológiák esetében[5].

8. fejezet

Összefoglalás és további fejlesztési irányok

8.1. Munka összefoglalása

A szakdolgozat a Media over QUIC (MoQ) protokoll architektúrájának és egyik alkalmazási lehetőségének bővítésével és vizsgálatával foglalkozott. A munka során sor került a protokoll elméleti hátterének az elemzésére, valamint a rendelkezésre álló implementációk közül az egyik képességeinek a tesztelésére.

A megszerzett ismeretekre alapozva egy kísérleti videóközvetítő rendszer kiegészült egy adaptív komponenssel, amely eltér a hagyományos szerveroldali vezérlésű modellektől, és a kliensoldali igény alapú vezérlést helyezi előtérbe. A rendszer kommunikációjának alapját egy saját tervezésű, JSON formátumú protokoll képezi. Ez a definíció lehetővé teszi a kliensek számára, hogy a videófolyam metaadatai alapján explicit átkódolási kéréseket fogalmazzanak meg, dinamikusan szabályozva a közvetítés paramétereit az átkódoló és a végpont között.

A rendszer komponenseinek implementációja C++ nyelven történt. A fejlesztés eredménye három modul: a forrásokat és a generált variánsokat kezelő Catalog Maker, az FFmpeg könyvtár alapú Transcoder, valamint a felhasználói interakciót biztosító Request Client. A tervezés folyamatában megmaradt a fő szempontok között a MoQ alapelvéinek az egyike, a skálázhatóság. Az elkészült architektúra bizonyítottan működőképes a kitűzött specifikáció elveit követve.

A funkcionális ellenőrzés mellett a rendszer egyéb tulajdonságait is megvizsgáltam. Ezek rávilágítottak arra, hogy a koncepció életképes és annak ellenére hogy a MoQ nem végleges szabvány és az implementációk is folyamatos változás és fejlesztés alatt állnak, már most alkalmas egy komplex rendszer alapjaként szolgálni. A bemutatott igényvezérelt architektúra működőképes és erőforrás-hatékony alternatívát jelent egy jövőben valószínűleg elterjedt protokoll használatával megvalósított videóközvetítő rendszer számára. Emellett a jelenlegi megvalósítás egy szűk keresztmetszetet is felfedett, csak annyira lehet hatékony a teljes folyamat, amennyire az átkódolást végző rendszerkomponens és annak kezelése engedi.

8.2. Továbbfejlesztési lehetőségek

A megvalósított rendszer egy működőképes prototípus, amely igazolta az igényvezérelt MoQ streaming megvalósíthatóságát. Azonban számos területen lehet a munkát folytatni, ezeket négy fő kategóriába soroltam.

8.2.1. Protokoll-szintű mélyítés és MoQ funkciók

Bár a jelenlegi implementáció a MoQ protokoll alapvető szállítási mechanizmusaira támaszkodik, a szabvány keretében számos egyéb funkció is fejlesztés alatt áll, amelyek integrálása tovább növelheti a rendszer képességeit. Ilyen például a QoS (Quality of Service) támogatás, amely lehetővé tenné a hálózati erőforrások hatékonyabb kihasználását és a szolgáltatás minőségének javítását. Emellett a jelenleg nagy figyelmet fektetnek a MoQ fejlesztése terén a biztonsági és authentikációs mechanizmusokra, amelyek beépítése növelné a rendszer megbízhatóságát és védelmét a potenciális támadásokkal szemben, illetve lehetőséget teremtene a felhasználók hitelesítése révén több szinten szabályozott hozzáférésre is.

8.2.2. Az átkódoló komponens optimalizálása

A teljesítménymérések rávilágítottak a szoftveres enkóder és a pufferkezelés korlátaira. A jelenlegi CPU-alapú FFmpeg megoldás helyett integrálni lehetne a hardveres gyorsítást. Emellett a kiértékelés során tapasztalt pufferelési probléma (lassú feldolgozás aktív bemenet esetén) a belső szálkezelés és a RingBuffer implementáció részletesebb kidolgozását indokolják. Egy eseményvezérelt, lock-free adatszerkezetre való áttérés megszüntetné a szálak közötti versengést, és egyenletesebb feldolgozási sebességet biztosítana.

8.2.3. Új átkódolási funkciók és szolgáltatások

A kidolgozott JSON alapú kérés-protokoll rugalmassága lehetővé teszi új, komplexebb kép- és hangfeldolgozási funkciók bevezetését a séma bővítésével. Lehetőség nyílhat speciális képjavító és szűrő algoritmusok implementálására, amelyek különösen gyenge minőségű forrásanyagok esetén hasznosak. Ilyen például a zajszűrés, a színcorrekció, vagy a villódzásmentesítés, amely a fényérzékeny epilepsziában szennedő nézők védelmét szolgálja.

A rendszer funkcionalitása tovább bővíthető gépi tanuláson alapuló, generatív AI modulok integrálásával. Ezek segítségével megvalósítható lenne a videótartalom valós idejű elemzése automatikus feliratgeneráláshoz, villódzásmentesítéshez, vagy akár a hangsáv azonnali fordítása és szinkronizálása más nyelvekre. Emellett a támogatott kodekek körének kiterjesztése (pl. AV1, VP9) lehetővé tenné a kliensek számára, hogy a H.264 mellett sávszélesség-takarékosabb formátumokat igényeljenek, optimalizálva az adatátvitelt a hálózati körülményekhez.

8.2.4. Skálázhatóság és Orkesztráció

A jelenlegi architektúrában az átkódoló egy önálló node-ként működik. A rendszer ipari környezetbe való emelése a skálázhatóság megoldását igényli.

A továbbfejlesztés irányába egy konténer-orkesztrációs rendszerrel (pl. Kubernetes) való integráció. Ebben a modellben szükséges lenne egyrészt egy pontos átkódoló szinkronizációs mechanizmus kidolgozása, amely biztosítja az átkódolók közötti szervezett terhelés-elosztást és kérésfeldolgozást. Másrészt egy dinamikus erőforrás-kezelő réteg bevezetése is indokolt ebben az esetben, amely a hálózati terhelés és a kliensigények alapján automatikusan skálázza az átkódoló példányok számát, optimalizálva ezzel a teljesítményt.

Köszönetnyilvánítás

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Irodalomjegyzék

- [1] Cloudflare: moq-rs: Media over QUIC in Rust. <https://github.com/cloudflare/moq-rs>, 2024. GitHub repository, Elérés ideje: 2025-12-03.
- [2] Luke Curley: moq: Media over QUIC library in Rust and TypeScript. <https://github.com/kixelated/moq>, 2024. GitHub repository, Elérés ideje: 2025-12-03.
- [3] FFmpeg Developers: FFmpeg Documentation. <https://ffmpeg.org/ffmpeg.html>, 2025. december. Accessed: 2025-12-07.
- [4] Mathis Engelbart: moqtransport: Media over QUIC Transport Implementation in Go. <https://github.com/mengelbart/moqtransport>, 2024. GitHub repository, Elérés ideje: 2025-12-03.
- [5] Gcore: How low latency streaming works, n.d. URL <https://gcore.com/docs/streaming/live-streaming/how-low-latency-streaming-works>. Elérés ideje: 2025-12-07.
- [6] ISO/IEC JTC 1/SC 29: ISO/IEC 23000-19:2024 Information technology – Multimedia application format (MPEG-A) – Part 19: Common media application format (CMAF) for segmented media. <https://webstore.iec.ch/publication/98515>, 2024. február. Third edition. Abstract: This document specifies the CMAF multimedia format, which contains segmented media objects optimized for streaming delivery and decoding on end-user devices in adaptive multimedia presentations. It defines a standard container for audio/visual content to be used in adaptive bitrate streaming (HLS/-DASH) and low-latency applications.
- [7] Jana Iyengar – Martin Thomson: QUIC: A UDP-Based Multiplexed and Secure Transport. 9000. Jelentés, 2021. május.
URL <https://www.rfc-editor.org/info/rfc9000>.
- [8] Will Law – Luke Curley – Victor Vasiliev – Suhas Nandakumar – Kirill Pugin: WARP Streaming Format. draft-ietf-moq-warp-01. Internet-Draft, 2025. július, Internet Engineering Task Force.
URL <https://datatracker.ietf.org/doc/draft-ietf-moq-warp/01/>. Work in Progress.
- [9] Niels Lohmann: JSON for modern c++.
URL <https://github.com/nlohmann/json>. Elérés ideje: 2025-12-07.
- [10] Lorenzo Miniero: Getting Media Over QUIC (MoQ) and WebRTC to like each other | Meetecho Blog, 2024. szeptember.
URL <https://www.meetecho.com/blog/moq-webrtc/>.

- [11] Uma Nair: Common media application format (cmaf): What it is and how it works. <https://www.gumlet.com/learn/what-is-cmaf/>, 2024. szeptember. Updated: 2024-09-03, Elérés ideje: 2025-11-13.
- [12] Suhas Nandakumar – Cullen Fluffy Jennings: Media Over QUIC Media and Security Architecture. draft-nandakumar-moq-arch-00. Internet-Draft, 2023. március, Internet Engineering Task Force. URL <https://datatracker.ietf.org/doc/draft-nandakumar-moq-arch/00/>. Work in Progress.
- [13] Suhas Nandakumar – Victor Vasiliev – Ian Swett – Alan Frindell: Media over QUIC Transport. draft-ietf-moq-transport-15. Internet-Draft, 2025. október, Internet Engineering Task Force. URL <https://datatracker.ietf.org/doc/draft-ietf-moq-transport/15/>. Work in Progress.
- [14] QuicR Project: LibQuicR: QuicR Library Implementation. <https://github.com/Quicr/libquicr>, 2024. GitHub repository, Elérés ideje: 2025-12-03.
- [15] Mo Zanaty – Suhas Nandakumar – Peter Thatcher: Low Overhead Media Container. draft-ietf-moq-loc-01. Internet-Draft, 2025. július, Internet Engineering Task Force. URL <https://datatracker.ietf.org/doc/draft-ietf-moq-loc/01/>. Work in Progress.

Függelék

F.1. Projekt GitHub repozitóriuma

A dolgozatban bemutatott rendszer forráskódja elérhető a következő GitHub tárolóban: Schweitzee/libquicr

F.2. Request JSON séma

Az átkódolás kérés JSON sémája megtalálható a következő GitHub tárolóban: Schweitzee/libquicr - transcode branch, dev/request_scheme.json