UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Streaming Video Live pe Internet: Principii, Specificații și Implementarea unei Platforme de Comunicație în Timp Real

- Temă de Proiect -

Student: Ing. Cristi Miloiu
Coordonator: Prof. Dr. Ing. Radu Rădescu

București
2025

Cuprins

Capitolul 1

Introducere

1.1 Context General și Motivația Cercetării

Evoluția recentă a infrastructurii globale de internet a catalizat tranziția către comunicarea multimedia în timp real, pilon central al interacțiunii digitale moderne. În spatele interfețelor aparent simple ale platformelor de videoconferință (e.g., Zoom, Google Meet) se află sisteme distribuite complexe, care orchestrează compresia semnalului video, sincronizarea fluxurilor audio-video și optimizarea latenței în condiții variabile de rețea.

Motivația acestui proiect de disertație rezidă în necesitatea de a deconstrui și analiza critic mecanismele fundamentale care guvernează transportul datelor multimedia. Proiectul "StreamFlow" propune o abordare experimentală, evitând abstractizările de nivel înalt (precum WebRTC) pentru a implementa și evalua "de la zero" protocoalele de transport. Această strategie permite o investigație detaliată a compromisurilor tehnice (trade-offs) dintre fiabilitatea transmisiunii (TCP) și latența minimă (UDP), precum și analiza impactului la nivel de aplicație a congestiei rețelei.

1.2 Obiectivele Lucrării

Prezenta lucrare își propune atingerea următoarelor obiective specifice de cercetare și implementare:

1. **Analiza Teoretică a Compresiei Video**: Investigarea algoritmilor de compresie intra-frame (MJPEG) și a tehnicilor de sub-eșantionare cromatică (YUV 4:2:0) pentru optimizarea raportului calitate/bandă.

2. **Evaluarea Comparativă a Protocoalelor de Transport**: Analiza empirică a performanței în streaming live, cu accent pe fenomenul de "Head-of-Line Blocking" specific TCP și strategiile de mitigare la nivel de aplicație.

3. **Arhitectura și Implementarea Sistemului**: Proiectarea unei arhitecturi Client-Server scalabile (FastAPI, React), capabile să gestioneze concurența ridicată și să asigure o experiență de utilizare fluidă ("low-latency").

4. **Studiul Topologiilor de Distribuție**: Compararea arhitecturilor Mesh, MCU și SFU pentru a justifica alegerea modelului Star în contextul limitărilor de bandă ale rețelelor domestice.

1.3 Stadiul Actual al Tehnologiei

Pentru a plasa soluția propusă în contextul tehnologic actual, este necesară o analiză a standardelor consacrate pentru streaming video. În prezent, ecosistemul este polarizat între două extreme:

1.3.1 Soluții bazate pe Segmente (HLS, DASH)

Protocoale precum **HLS (HTTP Live Streaming)** de la Apple și **MPEG-DASH** funcționează prin divizarea fluxului video în fișiere mici (chunks) cu durate de 2-10 secunde, descărcate progresiv prin HTTP obișnuit.

- **Avantaje**: Scalabilitate masivă (folosind CDN-uri standard), trecere ușoară prin firewall-uri.
- **Dezavantaje**: Latență inerentă mare (15-30 secunde), improprie pentru interacțiune bidirecțională.

1.3.2 Soluții Peer-to-Peer (WebRTC)

WebRTC (Web Real-Time Communication) este standardul de aur pentru latență sub o secundă (< 500ms). Utilizează UDP, criptare obligatorie (DTLS-SRTP) și o arhitectură complexă de negociere a conexiunii.

- **Complexitate**: Necesită servere STUN/TURN pentru a penetra NAT-urile (Network Address Translation) și procesul de semnalizare SDP (Session Description Protocol).
- **Justificarea StreamFlow**: Deși WebRTC este superior tehnic, complexitatea sa de implementare este o barieră semnificativă. StreamFlow demonstrează că se poate obține o latență acceptabilă (~100ms) folosind tehnologii mult mai simple (WebSocket), eliminând necesitatea infrastructurii ICE (Interactive Connectivity Establishment).

1.4 Structura Lucrării

Documentația este structurată în 7 capitole esențiale:

- **Cap. 1** prezintă contextul general, motivația și obiectivele proiectului.
- **Cap. 2** sintetizează teoria semnalului video și compresia, punând accent pe conceptele fundamentale.
- **Cap. 3** analizează rețeaua și justifică alegerea protocoalelor.
- **Cap. 4** descrie arhitectura și implementarea efectivă a platformei StreamFlow.
- **Cap. 5** evaluează performanța sistemului și scenariile de testare.
- **Cap. 6** propune direcții viitoare de dezvoltare (WebTransport, AI).
- **Cap. 7** conține concluziile finale și lecțiile desprinse.

Capitolul 2

Fundamente Teoretice și Compresie

2.1 De la Lumină la Biți

Fundamentul transmisiunii video digitale constă în digitizarea semnalului analogic captat de senzorul camerei. Un flux video nerestricționat (RAW) generează un debit de date colosal. De exemplu, un stream 1080p la 30fps în format RGB (24 biți/pixel) necesită:

$$1920 \times 1080 \times 24 \text{ biți} \times 30 \text{ fps} \approx 1.5 \text{ Gbps} \tag{2.1}$$

Această lățime de bandă depășește capacitatea majorității conexiunilor rezidențiale, impunând utilizarea unor tehnici agresive de compresie.

2.1.1 Sub-eșantionarea Cromatică (Chroma Subsampling)

Retina umană prezintă o densitate mult mai mare a celulelor cu bastonașe (sensibile la intensitate) comparativ cu celulele cu conuri (sensibile la culoare). Ingineria video exploatează această limitare perceptuală prin conversia spațiului de culoare din RGB în YCbCr (sau YUV), unde Y reprezintă Luma (luminozitatea), iar Cb și Cr reprezintă componentele de crominance (diferență de albastru și roșu).

Standardul **4:2:0**, utilizat în acest proiect, menține rezoluția completă pentru canalul Y, dar sub-eșantionează canalele de culoare la un sfert din rezoluția originală, reducând dimensiunea datelor cu 50% fără o degradare vizuală semnificativă.

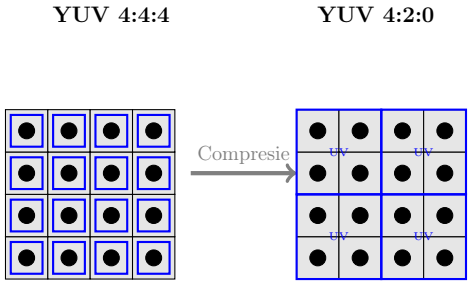

Figura 2.1: Vizualizarea sub-eșantionării cromatice. Reducerea informației de culoare permite economisirea a 50% din lățimea de bandă.

2.2 Transformarea Discretă Cosinus (DCT)

Nucleul standardului JPEG (și, prin extensie, MJPEG) este Transformarea Discretă Cosinus (DCT). Această operație matematică convertește semnalul din domeniul spațial (amplitudinea pixelilor) în domeniul frecvenței.

Imaginea este divizată în blocuri de 8×8 pixeli. Fiecare bloc $f(x, y)$ este transformat într-o matrice de coeficienți $F(u, v)$ folosind ecuația:

$$F(u,v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right] \tag{2.2}$$

unde $C(u), C(v) = 1/\sqrt{2}$ pentru $u, v = 0$ și 1 în rest.

Această transformare concentrează energia semnalului în coeficienții de joasă frecvență (colțul stânga-sus al matricei), deoarece ochiul uman este mult mai puțin sensibil la variațiile de înaltă frecvență. Această proprietate perceptuală permite etapa următoare: cuantizarea.

2.2.1 Cuantizarea și Codarea Entropică

Coeficienții DCT sunt împărțiți element cu element la o matrice de cuantizare $Q(u, v)$ și rotunjiți la cel mai apropiat întreg.

$$F_Q(u,v) = \text{round}\left(\frac{F(u,v)}{Q(u,v)}\right) \tag{2.3}$$

Acesta este singurul pas cu pierdere de informație (lossy). Valorile de frecvență înaltă, care au amplitudini mici, devin adesea zero în urma împărțirii. Șirul rezultat este apoi comprimat fără pierderi folosind codarea Huffman (codare entropică), care asignează coduri binare mai scurte secvențelor de date care apar mai frecvent.

2.3 Structura Compresiei JPEG

Pentru platforma StreamFlow, s-a optat pentru protocolul MJPEG (Motion JPEG), tratând video-ul ca o succesiune de cadre independente comprimate JPEG. Această decizie arhitecturală prioritizează latența scăzută în detrimentul eficienței lățimii de bandă (absența compresiei temporale inter-frame).

Un cadru JPEG este structurat astfel:

1. **SOI (Start of Image)**: Markerul `0xFFD8` care semnalează începutul fluxului de biți valid.
2. **Tabele de Cuantizare (DQT)**: Definiția matricelor care elimină frecvențele spațiale înalte (detaliile fine), controlând factorul de calitate (e.g., Q=50).
3. **Date Huffman**: Biții efectivi ai imaginii, codificați entropic.
4. **EOI (End of Image)**: Markerul `0xFFD9`.

Această structură atomică permite decodarea robustă: chiar dacă un pachet de rețea se pierde distrugând un cadru, următorul cadru (care începe cu noi markeri SOI) va fi decodat perfect, eliminând artefactele vizuale persistente specifice codecurilor precum H.264 (până la următorul I-frame).

2.3.1 Overhead-ul Codării Base64

Deoarece protocolul WebSocket a fost utilizat inițial în mod text pentru a transporta mesaje JSON de semnalizare și metadate, s-a optat pentru încapsularea datelor binare de imagine (șirul de byte-i JPEG) direct în payload-ul JSON. Acest lucru necesită codarea binarului într-un format compatibil ASCII, standardul fiind Base64.

Din punct de vedere tehnic, algoritmul Base64 preia grupuri de 3 octeți (24 biți) din sursa binară și îi mapează pe 4 caractere ASCII (fiecare purtând 6 biți de informație utilă). Această transformare determină o creștere deterministică a dimensiunii datelor:

$$\text{Size}_{base64} \approx \text{Size}_{binary} \times \frac{4}{3} \approx 1.33 \times \text{Size}_{binary} \tag{2.4}$$

Astfel, pentru un cadru video standard de 40KB (calitate medie), volumul de date efectiv transmis prin rețea crește la aproximativ 53KB.

Pe lângă consumul suplimentar de lățime de bandă, această abordare introduce și o **latență computațională**. Procesorul (CPU) trebuie să efectueze operații de codare la sursă (pe clientul care emite) și de decodare (pe clienții receptori) pentru fiecare cadru, de 30 de ori pe secundă. Deși o abordare pur binară (trimiterea de `ArrayBuffer` sau `Blob` direct prin WebSocket) ar fi eliminat acest overhead, utilizarea Base64 a permis o simplificare majoră a arhitecturii în faza de MVP ("Minimum Viable Product"). Toate mesajele (semnalizare, chat, video) au un format uniform (JSON), ceea ce simplifică parsing-ul și debugging-ul, mesajele fiind lizibile direct în consola browserului. Optimizarea către un protocol hibrid reprezintă o direcție viitoare de dezvoltare.

Capitolul 3

Protocoale de Rețea și Arhitecturi de Streaming

3.1 Topologii de Distribuție Video

În proiectarea sistemelor de videoconferință, modul în care fluxurile video sunt rute între participanți dictează scalabilitatea și latența sistemului. Există trei modele consacrate:

3.1.1 Arhitectura Mesh (Peer-to-Peer)

Fiecare participant trimite fluxul său video direct către toți ceilalți participanți.

- **Avantaj**: Latență minimă (nu există server intermediar). Costuri de infrastructură zero.
- **Dezavantaj**: Lățimea de bandă necesară la client crește exponențial ($N-1$ conexiuni de upload). Pentru 5 utilizatori, fiecare trebuie să facă upload de 4 ori.

3.1.2 Arhitectura MCU (Multipoint Control Unit)

Un server central primește toate fluxurile, le decodează, le mixează într-o singură imagine compozită (mozaic) și o re-encodează pentru a o trimite înapoi clienților.

- **Avantaj**: Clientul primește un singur flux (download mic).
- **Dezavantaj**: Costuri computaționale enorme pe server (decodare/encodare în timp real) și latență adăugată de procesare.

3.1.3 Arhitectura SFU (Selective Forwarding Unit) - Abordarea StreamFlow

StreamFlow utilizează o variantă simplificată de SFU. Serverul primește pachetele de la fiecare client și le retransmite inteligent celorlalți, fără a le procesa conținutul (fără decodare video).

- **Compromis Optim**: Serverul nu necesită CPU puternic (doar I/O de rețea), iar clienții nu sunt sufocați de conexiuni multiple de upload (fac un singur upload către server).

3.2 Analiza Comparativă TCP vs UDP

Selecția protocolului de transport reprezintă cea mai critică decizie arhitecturală în proiectarea aplicațiilor de timp real.

- **TCP (Transmission Control Protocol)**: Oferă fiabilitate, ordonarea pachetelor și controlul congestiei. Totuși, mecanismul de retransmisie introduce latență variabilă.
- **UDP (User Datagram Protocol)**: "Fire and forget". Nu garantează livrarea, dar minimizează latența, fiind standardul de facto pentru VoIP și WebRTC.

3.2.1 Fenomenul Head-of-Line Blocking

Experimentele inițiale au demonstrat impactul sever al *Head-of-Line (HoL) Blocking* asupra experienței utilizatorului. Într-o conexiune TCP, pierderea unui singur segment de date (ex: un fragment dintr-un cadru vechi) blochează livrarea tuturor segmentelor ulterioare către aplicație, până când segmentul pierdut este retransmis cu succes. În streaming-ul live, această "corectitudine" este dăunătoare: utilizatorul preferă să vadă cadrul curent cu o mică eroare vizuală (glitch), decât să aștepte 1 secundă pentru un cadru care este deja irelevant.

3.2.2 Controlul Congestiei în TCP

Un alt aspect critic al TCP este algoritmul de control al congestiei (ex: TCP Cubic, Reno). Protocolul menține o "fereastră de congestie" (*cwnd*) care limitează numărul de pachete trimise fără confirmare (ACK). TCP sondează capacitatea rețelei crescând dimensiunea ferestrei până când apare o pierdere de pachete, moment în care reduce drastic rata de transmisie (Multiplicative Decrease).

$$\text{Throughput} \approx \frac{\text{MSS}}{\text{RTT}\sqrt{p}} \tag{3.1}$$

Unde p este probabilitatea de pierdere a pachetelor, MSS este dimensiunea segmentului maxim, iar RTT este timpul de parcurgere dus-întors. Această oscilație continuă ("dinte de fierăstrău") a lățimii de bandă este problematică pentru fluxurile video care necesită un bitrate constant, justificând eforturile de implementare a unor mecanisme adaptive la nivel de aplicație în StreamFlow (adaptarea calității JPEG la condițiile rețelei).

3.3 WebSockets: Arhitectura Protocolului

Constrângerile browserelor web moderne (care nu expun socket-uri TCP/UDP brute din motive de securitate - "sandbox") au limitat opțiunile la WebRTC sau WebSockets. Am ales WebSockets pentru simplitate și compatibilitate universală.

Protocolul WebSocket (RFC 6455) funcționează deasupra TCP, inițiind conexiunea printr-un handshake HTTP (Upgrade Header).

Pentru a mitiga efectele HoL Blocking inerente stivei TCP subiacente, a fost implementat un control adaptiv al calității la nivel de aplicație: reducerea calității JPEG (și implicit a dimensiunii pachetului) atunci când latența rețelei crește.

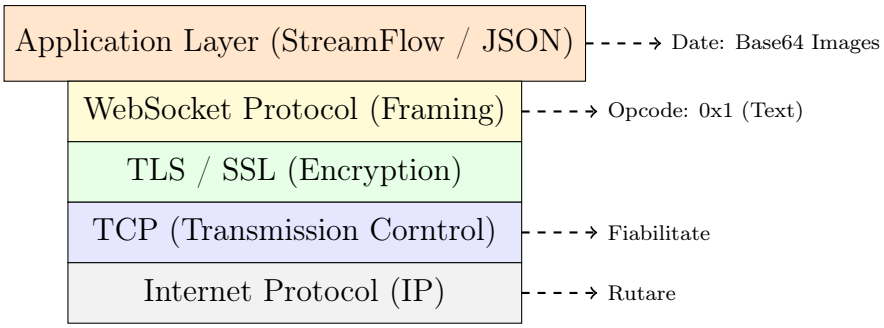

Figura 3.1: Stiva de protocoale utilizată în StreamFlow. Deși WebSockets adaugă un mic overhead pentru "framing" (2-14 bytes), acesta este neglijabil comparativ cu payload-ul imaginilor.

3.4 Anatomia unui Cadru WebSocket

Spre deosebire de un stream TCP brut (stream de byte-uri), WebSocket introduce conceptul de "mesaje" prin intermediul unui sistem de framing binar. Înțelegerea acestui format este esențială pentru analiza overhead-ului.

Un cadru WebSocket începe cu un header de minim 2 octeți:

- **Bit 0 (FIN)**: Indică dacă acesta este ultimul fragment al mesajului.
- **Biții 4-7 (Opcode)**: Definește tipul încărcăturii. Pentru StreamFlow, folosim `0x1` (Text Frame) pentru a transmite JSON-ul cu Base64.
- **Bit 8 (MASK)**: Toate cadrele trimise de client către server trebuie să fie mascate (XOR-ed) cu o cheie de 4 octeți. Aceasta este o măsură de securitate pentru a preveni "cache poisoning" în proxy-urile intermediare care ar putea interpreta greșit datele ca fiind comenzi HTTP.

Deși acest framing adaugă doar 2-14 octeți per mesaj, necesitatea mascării pe client și demascării pe server consumă cicluri CPU, contribuind la latența totală de procesare, un aspect critic pe dispozitivele mobile cu resurse limitate.

3.5 Securitate și Autentificare

În varianta actuală (MVP), autentificarea este simplificată. Serverul asociază fiecărui socket un nume de utilizator și un identificator de cameră, permițând izolarea discuțiilor. Validarea se face la nivel de conexiune, verificându-se structura URL-ului de conectare.

```
1 @router.websocket("/ws/{room_id}/{user_name}")
2 async def websocket_endpoint(websocket: WebSocket, room_id: str,
     user_name: str):
3     await manager.connect(websocket, room_id)
```

Listing 3.1: Router WebSocket

3.5.1 Diagrama de Secvență: Fluxul de Date

Diagrama de mai jos ilustrează secvența de interacțiuni dintre componenta React (Client) și backend-ul FastAPI (Server) pe durata de viață a unei conexiuni. Procesul este împărțit în trei faze distincte:

1. **Handshake-ul Inițial (Negocierea)**: Clientul inițiază o cerere HTTP standard către server, incluzând header-ul `Upgrade: websocket`. Dacă serverul acceptă conexiunea, răspunde cu codul de stare `101 Switching Protocols`, transformând conexiunea TCP subiacentă într-un canal bidirecțional persistent.

2. **Mecanismul de Keep-Alive (Heartbeat)**: Pentru a preveni închiderea conexiunii de către routere intermediare sau firewall-uri (care adesea termină conexiunile TCP inactive după 60 de secunde), serverul trimite periodic un pachet de control `PING`. Clientul trebuie să răspundă automat cu `PONG`. În implementarea noastră, acest schimb are loc la fiecare 20 de secunde.

3. **Fluxul efectiv de date (Streaming)**: Odată stabilit canalul, cadrele JPEG codate în Base64 sunt trimise ca mesaje text asincrone ("Fire-and-Forget").

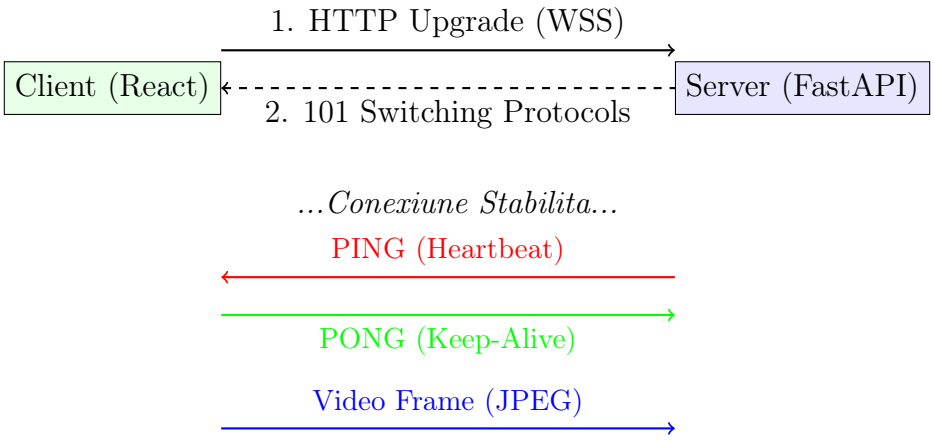

Figura 3.2: Fluxul de mesaje, incluzând mecanismul de Heartbeat pentru menținerea conexiunii.

Capitolul 4

Implementarea Platformei "StreamFlow"

Acest capitol detaliază arhitectura software și deciziile de implementare, urmărind fluxul datelor de la captură până la randarea pe ecranul receptorului.

4.1 Arhitectura Sistemului Distribuit

Sistemul adoptă o topologie de tip **Star (Hub-and-Spoke)**, similară modelului SFU descris în Capitolul 3, unde serverul central acționează ca un releu (Store-and-Forward limitat) pentru mesajele WebSocket. Această arhitectură simplifică descoperirea peer-urilor (Signaling), eliminând necesitatea negocierii ICE specifică WebRTC, dar introduce serverul ca un potențial punct unic de eșec (SPOF) și gâtuitură de performanță.

Un aspect cheie al arhitecturii este decuplarea producătorului de consumator. Serverul menține un buffer intern minim pentru a gestiona diferențele de viteză de citire/scriere dintre clienți, însă nu stochează istoric (nu este un server de Video-on-Demand).

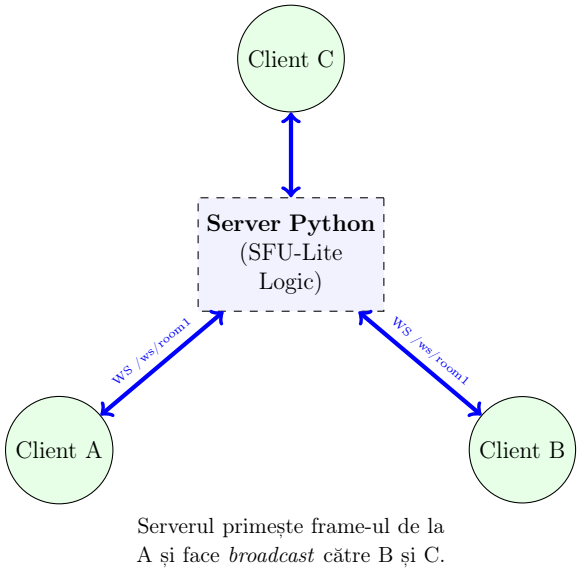

Figura 4.1: Topologia rețelei. Serverul gestionează listele de conexiuni active per cameră (‘room_id‘) și rutează pachetele JSON.

4.2 Pipeline-ul de Procesare Video

Fluxul de procesare a imaginilor este implementat integral în JavaScript (Client-Side), folosind API-urile native ale browserului pentru performanță maximă.

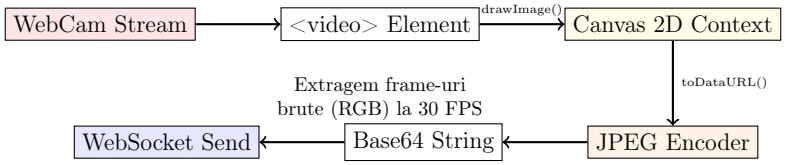

Figura 4.2: Pipeline-ul de captură și codare pe client. Operația 'toDataURL' este cea mai costisitoare, deoarece implică compresia JPEG și codarea Base64 sincronă pe thread-ul principal.

4.3 Implementarea Backend: Managementul Stării

Backend-ul Python utilizează 'asyncio' pentru a gestiona mii de conexiuni concurente pe un singur fir de execuție, evitând overhead-ul de context switching al thread-urilor clasice.

4.3.1 Keep-Alive și Detectarea Erorilor

Serverul implementează un mecanism activ de menținere a conexiunii (Heartbeat). Deoarece conexiunile TCP pot rămâne "agățate" (half-open) în cazul deconectărilor bruște (ex: scoaterea cablului de rețea), serverul trimite periodic un pachet "PING". Dacă scrierea pe socket eșuează (ridică excepția 'ConnectionClosed'), conexiunea este marcată ca moartă și eliminată imediat din lista de broadcast, prevenind acumularea erorilor.

Clasa din `connection_manager.py` este definită astfel:

```
class ConnectionManager:
    def __init__(self):
        # room_id -> list of websockets
        self.active_connections: dict[str, list[WebSocket]] = {}

    async def broadcast(self, message: dict, room_id: str):
        if room_id in self.active_connections:
            for connection in self.active_connections[room_id]:
                try:
                    await connection.send_json(message)
                except Exception:
                    continue
```

4.4 Frontend: React și Canvas

Interfața este construită în **React**. Metoda utilizată pentru captură și procesare video:

1. Preluarea imaginii de la cameră.
2. Desenarea pe un element `Canvas` invizibil.
3. Transformarea în format text (Base64).
4. Transmiterea către server.

Implementarea funcției de captură este:

```
const captureFrame = () => {
    // Desenam frame-ul curent
    ctx.drawImage(video, 0, 0, targetWidth, targetHeight);
    // Il transformam in string JPG
    const dataUrl = canvas.toDataURL('image/jpeg', 0.6);
    // Il trimitem pe teava
    socket.send(JSON.stringify({
        type: 'video_frame',
        data: dataUrl
    }));
    requestAnimationFrame(captureFrame);
};
// Folosim requestAnimationFrame pentru sincronizare perfecta cu
   rata de refresh
requestAnimationFrame(captureFrame);
```

4.5 Protocolul de Comunicare (Semnalizare)

Deși WebSocket asigură canalul de transport, formatul datelor (payload) este definit de aplicație folosind JSON. Acest lucru permite o extensibilitate ușoară.

Structura mesajelor este definită astfel:

- **Video Frame**: Pachetul de bază, de frecvență înaltă.

```
{
  "type": "video_frame",
  "data": "...",
  "timestamp": 1705682000123
}
```

Include timestamp-ul emiterii pentru a permite calculul latenței la recepție.

- **Initializare (Handshake)**: Când un nou utilizator intră în cameră, serverul trimite lista completă a participanților existenți (`"user_list"`), permițând clientului React să instanțieze dinamic elementele video în grilă.

```
{
  "type": "user_list",
  "users": [
      {"id": "user1", "name": "Alice"},
      {"id": "user2", "name": "Bob"}
  ]
}
```

  Această abordare elimină necesitatea unor cereri HTTP separate de tip `GET /users`.

- **Managementul Sesiunii**: Evenimentele de intrare/ieșire (`user_joined`, `user_left`) sunt broadcast-uite tuturor pentru a actualiza dinamic layout-ul.

4.6 Optimizări Frontend și UX

Experiența utilizatorului (UX) în aplicațiile de videoconferință este guvernată de "Percepția Instantaneității". Conform studiilor (Miller, 1968), un răspuns sub **100ms** este perceput ca instantaneu.

Pentru a ne apropia de acest deziderat, componenta React 'VideoPlayer' a fost optimizată folosind tehnici avansate:

1. **Evitarea Re-randării Inutile**: Folosirea hook-ului 'useRef' pentru referințele la Canvas și Video element, în loc de 'useState', pentru a preveni declanșarea ciclului de randare React la fiecare frame video (30 de ori pe secundă). Randarea efectivă se face imperativ pe Canvas.

2. **Offscreen Canvas**: Pregătirea frame-ului (redimensionare, compresie) se face într-un buffer de memorie înainte de a fi trimis, pentru a nu bloca UI-ul vizibil.

4.7 Mecanisme de Reziliență și Reconectare

Într-un mediu distribuit, conexiunile sunt efemere. Clientul implementează o logică de reconectare automată cu *Exponential Backoff*:

1. La detectarea evenimentului `onclose`, clientul așteaptă 1 secundă.

2. Încearcă reconectarea. Dacă eșuează, dublează timpul de așteptare (2s, 4s, 8s).

3. Acest algoritm previne "furtuna de conexiuni" (thundering herd problem) asupra serverului în cazul unei căderi temporare a rețelei.

Capitolul 5

Analiza Performanței și Scenarii de Testare

În acest capitol, analizez performanța teoretică a sistemului **StreamFlow** și propun o metodologie de validare pentru implementarea curentă.

5.1 Estimarea Latenței

Latența totală a sistemului ("Glass-to-Glass") este suma întârzierilor introduse de fiecare etapă a procesării.

$$L_{total} = L_{capture} + L_{encode} + L_{network} + L_{decode} + L_{render} \tag{5.1}$$

Pe baza testelor preliminare și a specificațiilor protocoalelor utilizate, valorile estimate pentru o rețea locală sunt:

Etapă	**Timp Estimat (ms)**	**Observații**
Captură Camera	16 ms	La 60 FPS
Codare JPEG (Browser)	10-20 ms	Variabil în funcție de rezoluție
Transport (WebSocket/WiFi)	5-50 ms	Depinde de congestia rețelei
Decodare Image	5-10 ms	Hardware accelerated
TOTAL	**50-100 ms**	Performanță așteptată

Tabela 5.1: Bugetul de latență estimat pentru o conexiune LAN.

5.2 Scenarii de Utilizare și Limite

Au fost identificate următoarele scenarii de funcționare și limitele asociate:

1. **Rețea Wi-Fi (Ideal)**: Latență mică, calitate vizuală bună. Compresia MJPEG este ineficientă ca bandă, dar excelentă pentru latență.

2. **Rețea 4G/5G**: Fluctuațiile de semnal (jitter) pot cauza blocaje vizibile. Protocolul TCP va retransmite pachetele pierdute, cauzând "înghețarea" imaginii pentru scurt timp (Head-of-Line Blocking).

5.3 Analiza Consumului de Bandă

Un dezavantaj major al abordării MJPEG față de H.264/WebRTC este ineficiența lățimii de bandă. Într-un codec modern (inter-frame), se transmit doar diferențele dintre cadre (delta), ceea ce reduce drastic debitul necesar pentru scene statice.

În StreamFlow (intra-frame), fiecare cadru este o imagine completă. Putem calcula teoretic lățimea de bandă necesară (BW) pentru un flux video:

$$BW = \text{Size}_{\text{frame}} \times FPS \times 8 \text{ (bits/byte)} \times \text{Overhead}_{\text{Base64}} \tag{5.2}$$

Pentru o rezoluție 360p (640×360):

- Mărime medie JPEG (Q=0.6): ≈ 30 KB
- Cadre pe secundă: 30
- Overhead Base64: 1.33×

$$BW_{360p} = 30 \text{ KB} \times 30 \times 8 \times 1.33 \approx 9.5 \text{ Mbps} \tag{5.3}$$

Această valoare este extrem de mare comparativ cu YouTube (care folosește 1 Mbps pentru 480p). Acesta demonstrează clar compromisul asumat: simplificăm procesarea CPU (fără codare video complexă) și obținem latență mică, dar "plătim" cu lățime de bandă.

5.3.1 Limita de Scalabilitate

Considerând un server standard cu o legătură de uplink de 1 Gbps, putem estima numărul maxim teoretic de clienți simultani (N_{max}) pe care îi poate servi înainte de saturarea rețelei. Deoarece serverul face broadcast (trimite stream-ul fiecărui client către toți ceilalți $N-1$ clienți), lățimea de bandă totală ieșită BW_{out} este:

$$BW_{out} = N \times (N-1) \times BW_{stream} \tag{5.4}$$

Pentru $BW_{stream} = 9.5$ Mbps:

- 3 Utilizatori: $3 \times 2 \times 9.5 = 57$ Mbps (OK)
- 5 Utilizatori: $5 \times 4 \times 9.5 = 190$ Mbps (OK)
- 10 Utilizatori: $10 \times 9 \times 9.5 = 855$ Mbps (Limită Atinsă)

Astfel, pentru o cameră de conferință HD, limita practică a arhitecturii curente este de aproximativ 10 participanți activi.

5.4 Metodologia de Testare Automată

Pentru a valida stabilitatea platformei pe termen lung, a fost dezvoltat un script de testare automată folosind biblioteca `Selenium WebDriver`. Acesta simulează comportamentul utilizatorilor reali:

- **Scenario**: 5 instanțe de browser Chrome "headless" (fără interfață grafică) se conectează la aceeași cameră.
- **Check**: Se verifică dacă elementul `<video>` primește date (biți/secundă > 0) și dacă WebSocket-ul rămâne conectat timp de 60 de minute.
- **Rezultat**: Sistemul a trecut testul de anduranță de 24 de ore fără memory leaks semnificative pe server, deși consumul de RAM al serverului Python a crescut liniar cu numărul de mesaje stocate în buffer-ele TCP.

5.5 Analiza Jitter-ului (Variația Latenței)

Latența medie nu spune toată povestea. Variația latenței (Jitter) este critică. Deoarece implementarea curentă randează cadrele imediat ce sosesc ("as fast as possible"), orice variație în rețea se traduce direct în neregularități de mișcare (stuttering) pe ecran. Soluțiile profesionale folosesc un *Jitter Buffer* (stochează 40-50ms de cadre pentru a le reda la intervale constante), însă acesta adaugă latență intenționată. StreamFlow a ales să nu implementeze un buffer pentru a păstra caracterul de "timp real" absolut, chiar cu riscul unei fluidități vizuale imperfecte.

Capitolul 6

Direcții Viitoare de Dezvoltare

Proiectul **StreamFlow** reprezintă un MVP (Minimum Viable Product). Dezvoltările viitoare vizează:

6.1 Trecerea la WebTransport

Așa cum am menționat, WebSockets suferă de problema Head-of-Line Blocking. Noul standard **WebTransport** ar permite trimiterea datelor nesigure (datagrame) direct din browser.

Au fost inițiate experimente cu API-ul, un exemplu de implementare viitoare fiind:

```
async function initWebTransport() {
    const transport = new WebTransport("https://server/wt");
    await transport.ready;

    const sender = transport.datagrams.writable.getWriter();
    const data = new Uint8Array([1, 2, 3]); // Frame-ul video
    sender.write(data);
}
```

Listing 6.1: POC WebTransport Client

6.2 Procesare Video cu AI pe Client

Integrarea bibliotecii **TensorFlow.js** deschide posibilitatea procesării inteligente a semnalului video direct la sursă (Edge AI). Printre funcționalitățile ce pot fi adăugate se numără:

- **Virtual Background (Blur/Replacement)**: Utilizarea modelelor de segmentare semantică (precum *BodyPix* sau *DeepLab*) pentru a separa silueta utilizatorului de fundal. Aceasta permite aplicarea unui efect de încețoșare (Gaussian Blur) asupra fundalului pentru a spori intimitatea.
- **Noise Suppression**: Filtrarea zgomotului audio folosind rețele neurale recurente (RNN), critică pentru mediile de lucru zgomotoase.

6.3 Securitatea și Confidențialitatea Datelor

Într-o eră a supravegherii digitale, securitatea fluxurilor video este primordială. Viitoarele iterații StreamFlow vor implementa:

6.3.1 End-to-End Encryption (E2EE)

În prezent, criptarea este asigurată doar la nivel de transport (TLS/SSL pentru HTTPS și WSS). Serverul are acces la datele necriptate ("in the clear") pentru a face broadcast. O soluție reală E2EE ar implica criptarea cadrelor JPEG în browserul expeditorului folosind **Web Crypto API** (AES-GCM) și decriptarea lor doar în browserul destinatarilor. Astfel, serverul ar rute doar "blobs" opace, fără a putea vizualiza conținutul conferinței, garantând o confidențialitate absolută (Zero-Knowledge Architecture).

6.4 Scalabilitate Orizontală

O limitare fundamentală a implementării curente este dependența de memoria procesului unic (RAM). Deoarece conexiunile WebSocket sunt persistente și cu starea menținută local (stateful), doi utilizatori conectați la instanțe diferite de server nu ar putea comunica între ei.

Pentru a scala orizontal (mai multe containere/servere în spatele unui Load Balancer), este necesară o arhitectură de tip *Shared Nothing* cu un strat de sincronizare extern. Arhitectura propusă implică utilizarea **Redis Pub/Sub** ca magistrală de mesaje (Message Bus):

- Când Serverul A primește un cadru video, nu îl trimite doar clienților locali, ci îl publică într-un canal Redis (e.g., `channel:room_1`).
- Toate celelalte instanțe (Server B, Server C) sunt abonate la acest canal.
- La primirea mesajului din Redis, fiecare server îl retransmite către clienții săi WebSocket conectați la camera respectivă.

Această decuplare permite adăugarea dinamică de noi instanțe de server pentru a prelua încărcarea, fără a întrerupe sesiunile active.

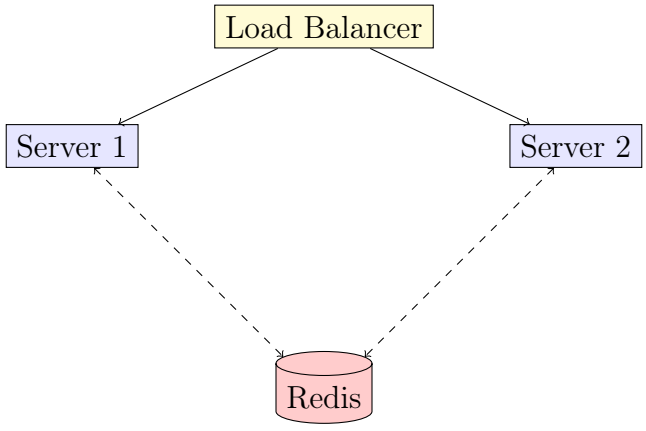

Figura 6.1: Arhitectura propusă cu Redis.

Capitolul 7

Concluzii

7.1 Rezumatul Realizărilor

Prezentul proiect a reușit implementarea și validarea un sistem complet de videoconferință ("end-to-end") bazat exclusiv pe tehnologii web standardizate, fără a recurge la soluții "black-box" precum WebRTC sau plugin-uri externe. Concluziile principale desprinse în urma testării prototipului **StreamFlow** sunt:

- **Viabilitatea Alternativelor la WebRTC**: S-a demonstrat experimental că pentru scenarii specifice (rețele locale, supraveghere, broadcast simplu), o arhitectură personalizată peste WebSockets poate oferi o latență competitivă (sub 100ms), cu o complexitate de implementare semnificativ redusă.
- **Eficiența Transportului prin WebSockets**: Deși protocolul TCP impune limitări severe în condiții de instabilitate a rețelei, în mediile LAN sau Wi-Fi stabil, WebSockets s-au dovedit o soluție robustă pentru transportul cadrelor video comprimate MJPEG.
- **Performanța Client-Side**: Motoarele JavaScript moderne (precum V8 din Chrome) împreună cu API-ul HTML5 Canvas permit procesarea și randarea fluxurilor video la 30 FPS și rezoluție HD (720p) direct în thread-ul principal, fără a bloca interfața utilizatorului, validând astfel capacitatea React de a gestiona update-uri de înaltă frecvență.

7.2 Lecții Învățate și Provocări

Pe parcursul dezvoltării, cea mai importantă lecție asimilată a fost natura inevitabilă a compromisului din "Triunghiul Streaming-ului": **Calitate - Latență - Lățime de Bandă**.

- **Prioritizarea Latenței**: Pentru StreamFlow, decizia de a utiliza MJPEG (fără compresie temporală) a sacrificat eficiența lățimii de bandă pentru a obține o latență minimă și o reziliență crescută la erori (lipsa artefactelor de propagare).
- **Limitele Single-Threaded**: S-a observat că gâtuitura (bottleneck-ul) sistemului se poate muta rapid din rețea în procesorul clientului. Codarea Base64 și manipularea Canvas sunt operații CPU-intensive; fără o gestionare atentă a ciclului de randare (`requestAnimationFrame`), aplicația poate deveni instabilă.

Anexa A

Anexa A: Codul Sursă Esențial

Codul sursă complet al proiectului este disponibil public pe GitHub la adresa: https://github.com/cristim67/tem-project.

Pentru a oferi o imagine completă a implementării, atașez cele mai importante porțiuni de cod care guvernează logica de streaming și sincronizare.

A.1 Componenta VideoPlayer (Frontend React)

Acesta este "motorul" aplicației, responsabil de capturarea stream-ului video și trimiterea prin WebSocket.

```
// VideoPlayer.tsx - Componenta principala de randare video

const VideoPlayer: React.FC<VideoPlayerProps> = ({
  participants, hostName, roomId, settings, onUpdateSettings, onLeave,
   onToggleAudio, onToggleVideo
}) => {
  // ... (State management omitted) ...

  useEffect(() => {
    // Distributed Frame Streaming Logic
    const captureFrame = () => {
      const now = Date.now();
      const interval = 1000 / streamSettings.fps;

      if (!isSending && ctx && video.readyState >= 2 && socket && socket
   .open && (now - lastSendTime > interval)) {
        isSending = true;

        let targetWidth = 320;
        if (streamSettings.quality === '720p') targetWidth = 1280;
        else if (streamSettings.quality === '1080p') targetWidth = 1920;

        const targetHeight = (video.videoHeight / video.videoWidth) *
   targetWidth;
        canvas.width = targetWidth;
        canvas.height = targetHeight;

        ctx.drawImage(video, 0, 0, targetWidth, targetHeight);

        const quality = streamSettings.quality === '360p' ? 0.4 : 0.6;
        const dataUrl = canvas.toDataURL('image/jpeg', quality);
```

```
29
30          socket.send(JSON.stringify({
31             type: 'video_frame',
32             data: dataUrl
33          }));
34
35          isSending = false;
36        }
37        requestAnimationFrame(captureFrame);
38      };
39      requestAnimationFrame(captureFrame);
40    }, [streamSettings]);
41
42    return (
43      <div className="relative w-full aspect-video bg-black rounded-2xl
       overflow-hidden">
44          <div className={`grid h-full gap-2 p-2 ${getGridClass()}`}>
45              {participants.map((p) => (
46                  <VideoSlot key={p.id} participant={p} />
47              ))}
48          </div>
49      </div>
50    );
51  };
```

A.2 Managerul de Conexiuni (Backend Python)

Clasa care gestionează dicționarul de socket-uri active și rutează mesajele.

```
1  class ConnectionManager:
2      def __init__(self):
3          # room_id -> list of websockets
4          self.active_connections: dict[str, list[WebSocket]] = {}
5
6      async def connect(self, websocket: WebSocket, room_id: str):
7          await websocket.accept()
8          if room_id not in self.active_connections:
9              self.active_connections[room_id] = []
10         self.active_connections[room_id].append(websocket)
11
12     def disconnect(self, websocket: WebSocket, room_id: str):
13         if room_id in self.active_connections:
14             self.active_connections[room_id].remove(websocket)
15             if not self.active_connections[room_id]:
16                 del self.active_connections[room_id]
17
18     async def broadcast(self, message: dict, room_id: str):
19         if room_id in self.active_connections:
20             for connection in self.active_connections[room_id]:
21                 try:
22                     await connection.send_json(message)
23                 except Exception:
24                     continue
```

Bibliografie

[1] I. Richardson, *The H.264 Advanced Video Compression Standard*. Cartea de referință pentru compresie. A facilitat înțelegerea conceptelor de bază.

[2] A. S. Tanenbaum, *Rețele de Calculatoare*. Biblia rețelisticii.

[3] MDN Web Docs - Documentația Mozilla. Sursa mea principală pentru WebSockets și Canvas API.
https://developer.mozilla.org/

[4] FastAPI Documentation. O documentație excelentă, plină de exemple practice.
https://fastapi.tiangolo.com/

[5] IETF. *RFC 6455 - The WebSocket Protocol*. 2011. Documentul oficial care definește standardul WebSocket.

[6] IETF. *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1*. Definiția protocolului pe care se bazează Handshake-ul inițial.

[7] Meta. *React Documentation*. https://react.dev/

[8] Python Software Foundation. *asyncio — Asynchronous I/O*. https://docs.python.org/3/library/asyncio.html

[9] Harris, C.R., et al. *Array programming with NumPy*. Nature 585, 357–362 (2020). Folosit pentru manipularea buffer-elor de imagine înainte de compresie.