

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

1	Introducere	5
1.1	Context General și Motivația Cercetării	5
1.2	Obiectivele Lucrării	5
1.3	Structura Lucrării	6
2	Fundamente Teoretice și Compresie	7
2.1	De la Lumină la Biți	7
2.1.1	Sub-eșantionarea Cromatică (Chroma Subsampling)	7
2.2	Structura Compresiei JPEG	8
2.2.1	Overhead-ul Codării Base64	8
3	Protocoale de Rețea și Transport	9
3.1	Analiza Comparativă TCP vs UDP	9
3.1.1	Fenomenul Head-of-Line Blocking	9
3.2	WebSockets: Arhitectura Protocolului	9
3.3	Securitate și Autentificare	10
3.3.1	Diagrama de Secvență: Fluxul de Date	10
4	Implementarea Platformei "StreamFlow"	11
4.1	Arhitectura Sistemului Distribuit	11
4.2	Pipeline-ul de Procesare Video	12
4.3	Implementarea Backend: Managementul Stării	12
4.3.1	Keep-Alive și Detectarea Erorilor	12
4.4	Frontend: React și Canvas	13
5	Analiza Performanței și Scenarii de Testare	15
5.1	Estimarea Latenței	15
5.2	Scenarii de Utilizare și Limite	15
6	Direcții Viitoare de Dezvoltare	17
6.1	Trecerea la WebTransport	17
6.2	Procesare Video cu AI pe Client	17
6.3	Scalabilitate Orizontală	18
7	Concluzii	19
7.1	Rezumatul Realizărilor	19
7.2	Lecții Învățate și Provocări	19

A	Anexa A: Codul Sursă Esențial	21
A.1	Componenta VideoPlayer (Frontend React)	21
A.2	Managerul de Conexiuni (Backend Python)	22

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").

1.3 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:

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

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 cromatică (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.

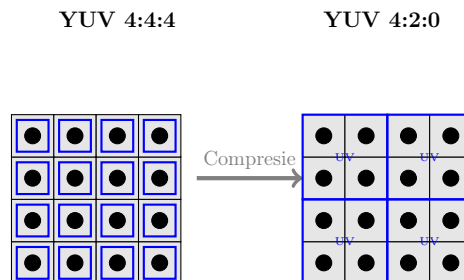


Figura 2.1: Vizualizarea sub-eșantionării cromatice. Reducerea informației de culoare permite economisirea a 50% din lățimea de bandă.

2.2 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).

2.2.1 Overhead-ul Codării Base64

Deoarece WebSockets transferă text în mod implicit (și pentru a simplifica manipularea în JSON), fluxul binar JPEG este codat în Base64 înainte de transmisie. Aceasta introduce un overhead semnificativ:

$$\text{Size}_{base64} \approx \text{Size}_{binary} \times \frac{4}{3} \approx 1.33 \times \text{Size}_{binary} \quad (2.2)$$

Pentru un cadru de 40KB, transmisia efectivă devine $\approx 53KB$. Deși ineficient, acest compromis a fost acceptat pentru a simplifica arhitectura MVP și a evita complexitatea manipulării `ArrayBuffer` și `Blobs` în faza inițială.

Capitolul 3

Protocoale de Rețea și Transport

3.1 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.

3.1.1 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 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.

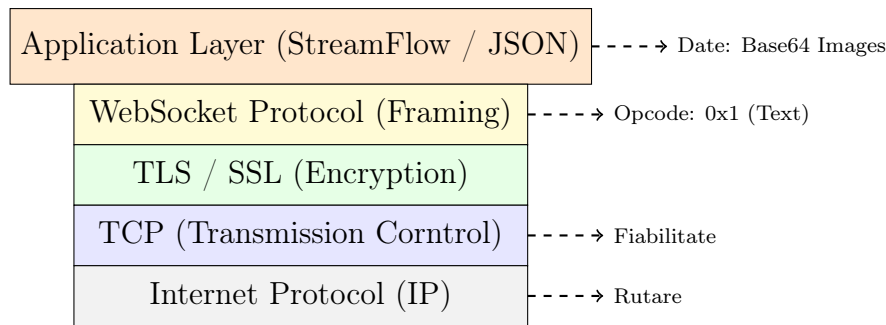


Figura 3.1: Stiva de protocoale utilizată în StreamFlow. Deși WebSockets adaugă un mic overhead pentru "framing" (2-14 bytes), acesta este neglijabil comparativ cu payload-ul imaginilor.

3.3 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,
3   user_name: str):
4     await manager.connect(websocket, room_id)

```

Listing 3.1: Router WebSocket

3.3.1 Diagrama de Secvență: Fluxul de Date

Pentru a clarifica modul în care funcționează conexiunea, iată diagrama simplificată.

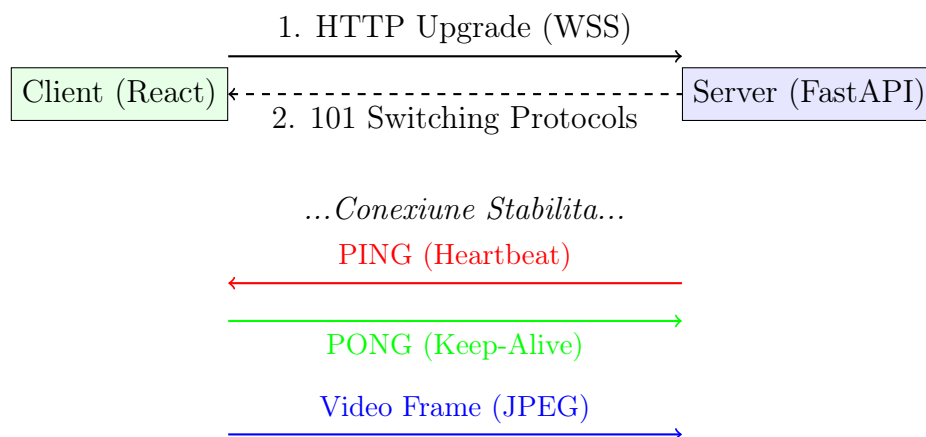


Figura 3.2: Fluxul de mesaje, incluzând mecanismul de Heartbeat pentru menținerea conexiunii.

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)**, 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.

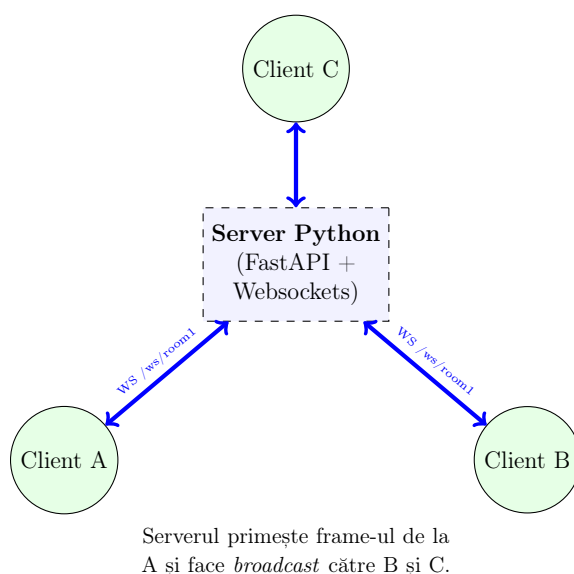


Figura 4.1: Topologia rețelei. Serverul gestionează listele de conexiuni active per cameră ('room_id') și rutează pachetele JSON.

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.

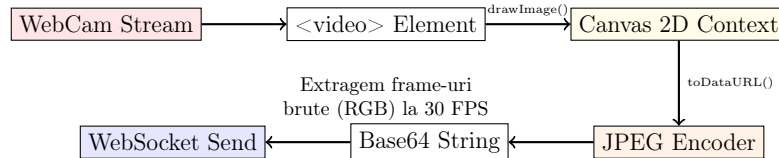


Figura 4.2: Pipeline-ul de captură și codare pe client. Operația 'toDataURL' este cea mai costisitoare, deoarece implică compresia JPEG și codarea Base64 sincronă pe thread-ul principal.

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:

```
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 broadcast(self, message: dict, room_id: str):
7         if room_id in self.active_connections:
8             for connection in self.active_connections[room_id]:
9                 try:
10                     await connection.send_json(message)
11                 except Exception:
12                     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:

```
1 const captureFrame = () => {
2   // Desenam frame-ul curent
3   ctx.drawImage(video, 0, 0, targetWidth, targetHeight);
4   // Il transformam in string JPG
5   const dataUrl = canvas.toDataURL('image/jpeg', 0.6);
6   // Il trimitem pe teava
7   socket.send(JSON.stringify({
8     type: 'video_frame',
9     data: dataUrl
10  }));
11   requestAnimationFrame(captureFrame);
12 };
13 // Folosim requestAnimationFrame pentru sincronizare perfecta cu
14 // rata de refresh
15 requestAnimationFrame(captureFrame);
```


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.

$$L_{total} = L_{capture} + L_{encode} + L_{network} + L_{decode} + L_{render} \quad (5.1)$$

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 inefficientă 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).

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 (datagramme) direct din browser.

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

```
1 async function initWebTransport() {  
2     const transport = new WebTransport("https://server/wt");  
3     await transport.ready;  
4  
5     const sender = transport.datagrams.writable.getWriter();  
6     const data = new Uint8Array([1, 2, 3]); // Frame-ul video  
7     sender.write(data);  
8 }
```

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 încetșare (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 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/servele 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.

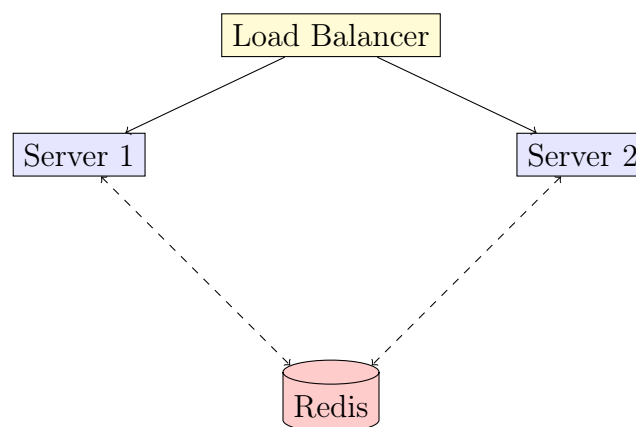


Figura 6.1: Arhitectura propusă cu Redis.

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 compimate 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.

```
1 // VideoPlayer.tsx - Componenta principala de randare video
2
3 const VideoPlayer: React.FC<VideoPlayerProps> = ({
4   participants, hostName, roomId, settings, onUpdateSettings, onLeave,
5   onToggleAudio, onToggleVideo
6 }) => {
7   // ... (State management omitted) ...
8
9   useEffect(() => {
10     // Distributed Frame Streaming Logic
11     const captureFrame = () => {
12       const now = Date.now();
13       const interval = 1000 / streamSettings.fps;
14
15       if (!isSending && ctx && video.readyState >= 2 && socket && socket
16         .open && (now - lastSendTime > interval)) {
17         isSending = true;
18
19         let targetWidth = 320;
20         if (streamSettings.quality === '720p') targetWidth = 1280;
21         else if (streamSettings.quality === '1080p') targetWidth = 1920;
22
23         const targetHeight = (video.videoHeight / video.videoWidth) *
24           targetWidth;
25         canvas.width = targetWidth;
26         canvas.height = targetHeight;
27
28         ctx.drawImage(video, 0, 0, targetWidth, targetHeight);
29
30         const quality = streamSettings.quality === '360p' ? 0.4 : 0.6;
31         const dataUrl = canvas.toDataURL('image/jpeg', quality);
```

```

29         socket.send(JSON.stringify({
30             type: 'video_frame',
31             data: dataUrl
32         }));
33
34         isSending = false;
35     }
36     requestAnimationFrame(captureFrame);
37 };
38 requestAnimationFrame(captureFrame);
39 }, [streamSettings]);
40
41
42 return (
43     <div className="relative w-full aspect-video bg-black rounded-2xl
44     overflow-hidden">
45         <div className={`grid h-full gap-2 p-2 ${getGridClass()}`>
46             {participants.map((p) => (
47                 <VideoSlot key={p.id} participant={p} />
48             ))}
49         </div>
50     </div>
51 );
52 };

```

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.