UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

DIPARTIMENTO DI SCIENZE
FISICHE, INFORMATICHE E MATEMATICHE

Corso di Laurea Magistrale in Informatica

Sistema per videoconferenze cifrate End to End basato sugli standard emergenti Sframe e MLS

Relatore:
Prof. Luca Ferretti

Candidato:
Danilo Bellocchio

Anno Accademico 2024–2025

Indice

1 Introduzione **1**

2 Stato dell'Arte e Background Tecnologico **5**

2.1 WebRTC e Sicurezza dei Flussi Multimediali 5

2.1.1 L'Approccio Ibrido al Livello di Trasporto: UDP vs TCP 6

2.1.2 Il Ruolo della Segnalazione in WebRTC 6

2.1.3 Topologie per Comunicazioni di Gruppo 8

2.1.4 Limiti di Sicurezza e Doppia Cifratura 8

2.1.5 Meccanismi di Intercettazione dei Flussi Multimediali 10

2.2 Protezione del Flusso Multimediale (SFrame) 11

2.3 Gestione delle Chiavi di Gruppo (MLS) 13

3 Progettazione **15**

3.1 Architettura Logica 15

3.1.1 Mappatura dei Componenti e Caratteristiche Funzionali 16

3.1.2 Dati Statici e Pre-condivisi 18

3.2 Protocolli di Comunicazione e Logica Ibrida 18

3.2.1 Protocollo di Inizializzazione della Stanza 20

3.2.2 Protocollo di Setup dell'Identità (Join Request) 20

3.2.3 Protocollo di Ingresso Utente e Sala d'Attesa Crittografica . . . 21

3.2.4 Protocollo di Rotazione delle Chiavi e Monitoraggio 22

3.2.5 Protocollo di Segnalazione WebRTC 22

3.2.6 Protocollo di Trasmissione Multimediale (SFrame) 22

4 Implementazione del Sistema **24**
4.1 Stack Tecnologico e Ambiente di Sviluppo 24
4.2 Il Core Crittografico e l'Integrazione WebAssembly 27
4.2.1 Gestore delle Identità (WasmMlsClient) 28
4.2.2 Gestore dell'Elaborazione Media (WasmPeer) 28
4.3 Architettura del Client Web e Organizzazione dei Moduli 29
4.3.1 Logica Applicativa del Client 30
4.4 Architettura Ibrida e Delivery Service E2EE 32
4.4.1 Il Delivery Service 33
4.4.2 Il Flusso Crittografico nel Client Web 33
4.4.3 Derivazione delle Chiavi di Trasporto e Calcolo del KID 35
4.5 Intercettazione dei Flussi Multimediali (Transform Streams) 36
4.5.1 Pipeline di Trasmissione (Sender) 36
4.5.2 Pipeline di Ricezione (Receiver) 37

5 Implementazione e discussione sperimentale **38**
5.1 Limiti di astrazione in WebRTC: l'incompatibilità con LiveKit 39
5.2 Persistenza dello stato in WebAssembly 40
5.3 Valutazione Qualitativa e Quantitativa: L'Impatto dell'Architettura E2EE 41
5.3.1 Metodologia di Misurazione: Il Control Plane 42
5.3.2 Valutazione Qualitativa: Il Media Plane 42
5.3.3 Analisi dei Tempi di OpenMLS 43
5.4 Maturità degli Standard e Scelte Implementative: SFrame vs MLS . . . 44

6 Conclusioni e Sviluppi Futuri **46**

Capitolo 1

Introduzione

Negli ultimi anni, le modalità di interazione e collaborazione a distanza hanno subito una profonda trasformazione, rendendo i sistemi di *web conferencing* un elemento fondamentale della comunicazione quotidiana. Se fino a qualche tempo fa la videoconferenza era una tecnologia complessa e di nicchia, oggi è diventata uno strumento indispensabile. Questa evoluzione ha subito una forte accelerazione a seguito della recente emergenza sanitaria, che ha spinto l'adozione su larga scala di modelli come lo *smart working*, l'*e-learning* e la telemedicina.

Questa rapida crescita della domanda ha messo a dura prova le architetture di rete, evidenziando i limiti dei protocolli di *streaming* tradizionali. Inizialmente, la distribuzione video si basava su tecnologie proprietarie o pensate per reti private, come RTSP (*Real Time Streaming Protocol*). Per la distribuzione su larga scala, il mercato si è poi orientato verso protocolli adattivi basati su HTTP, come HLS (*HTTP Live Streaming*) o lo standard aperto MPEG-DASH (*Dynamic Adaptive Streaming over HTTP*). Tuttavia, pur essendo ottimi per il *broadcasting* unidirezionale, questi protocolli introducono latenze incompatibili con la comunicazione bidirezionale in tempo reale necessaria per le videoconferenze.

Per superare questi limiti, si è affermato lo standard *Web Real-Time Communication* (WebRTC), un progetto *open-source* diventato il riferimento principale per lo scambio *peer-to-peer* di flussi multimediali a bassa latenza. Il successo di WebRTC dimostra come, affinché un nuovo protocollo diventi uno standard globale supportato nativamente dai

browser, sia necessario un rigoroso processo di validazione da parte di enti internazionali come il W3C e l'IETF. Questo percorso assicura che le nuove tecnologie risultino scalabili, interoperabili e, soprattutto, sicure.

La sicurezza informatica rappresenta oggi una delle sfide principali nel *web conferencing*. Per supportare chiamate di gruppo con numerosi partecipanti senza saturare la banda degli utenti, le piattaforme moderne si basano su server centrali chiamati *Selective Forwarding Unit* (SFU), che inoltrano selettivamente i flussi multimediali verso i dispositivi riceventi. Tuttavia, le implementazioni standard di WebRTC proteggono queste comunicazioni con un approccio *Hop-by-Hop*: la connessione sicura viene instaurata tra il singolo *client* e il server. Di conseguenza, l'infrastruttura centrale possiede le chiavi crittografiche della sessione e ha tecnicamente accesso in chiaro all'intero contenuto audio e video. Questo modello centralizzato richiede un elevato livello di fiducia nel fornitore del servizio e costituisce una potenziale vulnerabilità qualora si scambino informazioni sensibili o l'infrastruttura venga compromessa. L'esigenza di garantire una vera confidenzialità *End-to-End* (E2EE), in cui solo i legittimi partecipanti possiedono le chiavi, ha quindi spinto la ricerca verso nuovi paradigmi.

L'obiettivo di questa tesi è progettare e realizzare un'architettura per videoconferenze di gruppo che unisca sicurezza E2EE e la scalabilità dei sistemi centralizzati. A tale scopo, sono stati integrati due standard IETF: *Secure Frame* (SFrame) per il *media plane* e *Messaging Layer Security* (MLS) per il *control plane*. SFrame permette di slegare la cifratura dal livello di trasporto agendo a livello applicativo: il browser cifra il singolo fotogramma prima dell'invio, così l'infrastruttura riceve un *payload* già protetto. Questo consente al server di instradare i flussi in modo efficiente basandosi solo sugli *header* esterni, senza mai accedere ai dati in chiaro. Parallelamente, l'integrazione di MLS risolve il problema della scalabilità crittografica. Rispetto agli approcci classici, dove il costo di aggiornamento delle chiavi aumenta drasticamente con il numero di utenti, MLS utilizza una struttura ad albero che ottimizza il traffico di rete e garantisce proprietà di sicurezza avanzate.

Lo sviluppo del sistema ha richiesto l'adozione di soluzioni specifiche per eseguire le operazioni di cifratura direttamente sui *client*. L'intercettazione e la manipolazione dei fotogrammi video in tempo reale sono state realizzate sfruttando le interfacce native

del browser, ovvero le *WebRTC Encoded Transforms.* Infine, per non compromettere la fluidità della comunicazione, l'esecuzione dei complessi algoritmi crittografici è stata delegata a moduli ad alte prestazioni precompilati in *WebAssembly* (WASM), evitando di sovraccaricare il motore JavaScript.

Attualmente, il mercato offre diverse soluzioni per la sicurezza delle comunicazioni di gruppo, ma spesso con limitazioni architetturali. Piattaforme come Google Meet o Zoom proteggono i dati in transito, ma per impostazione predefinita decifrano i media sui propri server per abilitare funzionalità *cloud* (es. registrazione o sottotitoli). Sebbene forniscano opzioni E2EE o di *Client-Side Encryption*, queste si basano su ecosistemi chiusi, richiedono licenze *Enterprise* o l'uso di client proprietari, limitando l'interoperabilità. Anche evoluzioni recenti come lo standard MoQ (*Media over QUIC*) si concentrano sull'ottimizzazione del trasporto per ridurre la latenza, senza però affrontare la separazione crittografica a livello applicativo. L'obiettivo di questo lavoro, quindi, non è reimplementare un'applicazione esistente, ma verificare la fattibilità di un'architettura sicura, scalabile e basata esclusivamente su standard aperti IETF.

Il sistema realizzato è stato testato in un ambiente sperimentale, confermandone la fattibilità pratica e la capacità di proteggere efficacemente i flussi multimediali in tempo reale. Tuttavia, lo sviluppo ha richiesto il superamento di diverse sfide. Trattandosi di standard molto recenti, è stata necessaria un'attenta selezione tra i progetti *open-source* disponibili, poiché molte librerie risultavano ancora immature o incompatibili con l'ecosistema del browser. Adattare le soluzioni scelte per l'esecuzione in *WebAssembly* ha rappresentato una delle complessità maggiori del lavoro, insieme alla gestione asincrona dello stato crittografico e ai limiti di astrazione imposti da alcuni framework WebRTC. Seppur pienamente funzionante, il prototipo evidenzia quindi alcune difficoltà pratiche di integrazione che offrono spunti per sviluppi futuri.

L'elaborato è strutturato seguendo un percorso che va dall'analisi teorica alla validazione pratica. Il Capitolo 2 delinea il contesto tecnologico e i limiti di sicurezza di WebRTC, introducendo i fondamenti dei protocolli SFrame e MLS. Il Capitolo 3 descrive l'architettura logica e i protocolli di comunicazione progettati per gestire la stanza virtuale. Il Capitolo 4 illustra l'implementazione pratica, esaminando lo *stack* di sviluppo, l'integrazione del motore crittografico in *WebAssembly* e i meccanismi di

intercettazione dei fotogrammi. Nel Capitolo 5 vengono valutate le prestazioni del prototipo e le scelte ingegneristiche, analizzando nel dettaglio gli ostacoli affrontati durante lo sviluppo, come la persistenza dello stato e i limiti delle librerie attuali. Infine, il Capitolo 6 riassume i risultati ottenuti e delinea i possibili sviluppi futuri.

Capitolo 2

Stato dell'Arte e Background Tecnologico

Questo capitolo fornisce il contesto tecnologico e teorico necessario per comprendere le moderne architetture di videoconferenza. Vengono analizzati i principi base della comunicazione in tempo reale sul web, le vulnerabilità di sicurezza delle topologie attuali e gli standard emergenti necessari per garantire la confidenzialità delle comunicazioni.

2.1 WebRTC e Sicurezza dei Flussi Multimediali

WebRTC (*Web Real-Time Communication*) è la tecnologia alla base delle moderne applicazioni di videoconferenza via browser. Si tratta di uno standard aperto che permette a due o più dispositivi di trasmettere flussi audio e video in tempo reale direttamente dal browser, senza software o plugin aggiuntivi. Nella sua forma originaria, WebRTC stabilisce comunicazioni *Peer-to-Peer* (P2P) dirette tra i nodi. In questa topologia, lo scambio dei flussi multimediali avviene senza server intermediari ed è gestito dal protocollo di trasporto RTP (*Real-time Transport Protocol*). RTP è lo standard fondamentale per lo streaming di rete: incapsula i flussi audio e video in pacchetti, assegna loro marcatori temporali (*timestamp*) e numeri di sequenza, garantendo bassa latenza e il corretto ordine di riproduzione.

2.1.1 L'Approccio Ibrido al Livello di Trasporto: UDP vs TCP

Le architetture WebRTC adottano un approccio ibrido al livello di trasporto, separando la gestione dei flussi multimediali da quella dei dati di controllo.

Per la trasmissione di audio e video (il *Media Plane* gestito da RTP), WebRTC privilegia il protocollo **UDP** (*User Datagram Protocol*). Questa scelta è dettata dalla necessità di minimizzare la latenza: a differenza del TCP, UDP non prevede meccanismi di conferma di ricezione o ritrasmissione dei pacchetti. Ciò evita il fenomeno del blocco in testa (*Head-of-Line blocking*), che in una chiamata in diretta causerebbe ritardi o blocchi nella riproduzione (*stuttering*). L'implicazione principale di questa scelta è che UDP **non garantisce né la consegna certa né l'ordine di arrivo dei pacchetti** (fenomeno dell'*out-of-order delivery*).

Al contrario, per lo scambio dei messaggi di configurazione o crittografici (il *Control Plane*), il sistema si affida a protocolli basati su **TCP** (*Transmission Control Protocol*), come WSS (WebSocket Secure) o HTTPS. TCP garantisce una consegna affidabile e ordinata, requisito fondamentale affinché lo stato logico e le chiavi di sessione dei *client* rimangano sincronizzati.

Questa differenza fondamentale tra i due livelli di trasporto è la causa principale delle sfide di sincronizzazione che emergono quando si applica una cifratura *End-to-End* dinamica ai flussi multimediali in tempo reale.

2.1.2 Il Ruolo della Segnalazione in WebRTC

WebRTC, pur essendo progettato per la trasmissione multimediale diretta, non include un meccanismo per consentire ai nodi di localizzarsi sulla rete. Prima di scambiarsi flussi audio e video, i dispositivi devono identificare un percorso di rete e concordare i parametri tecnici della comunicazione.

Questa fase di negoziazione è detta **Segnalazione** (*Signaling*) e avviene tramite un server dedicato su protocolli di trasporto sicuri, tipicamente **WSS** (*WebSocket Secure*). Attraverso questo canale, i nodi utilizzano innanzitutto il framework **ICE** (*Interactive Connectivity Establishment*): un meccanismo di *NAT Traversal* che permette ai *client*

di scambiarsi i propri candidati di rete per determinare il percorso IP più efficiente attraverso firewall e NAT. Stabilita la connettività, i nodi si scambiano pacchetti di testo secondo lo standard **SDP** (*Session Description Protocol*) per negoziare le capacità multimediali (es. codec e risoluzioni). Completata la selezione del percorso ICE e la negoziazione SDP, i partecipanti possono instaurare il canale e avviare il trasferimento multimediale.

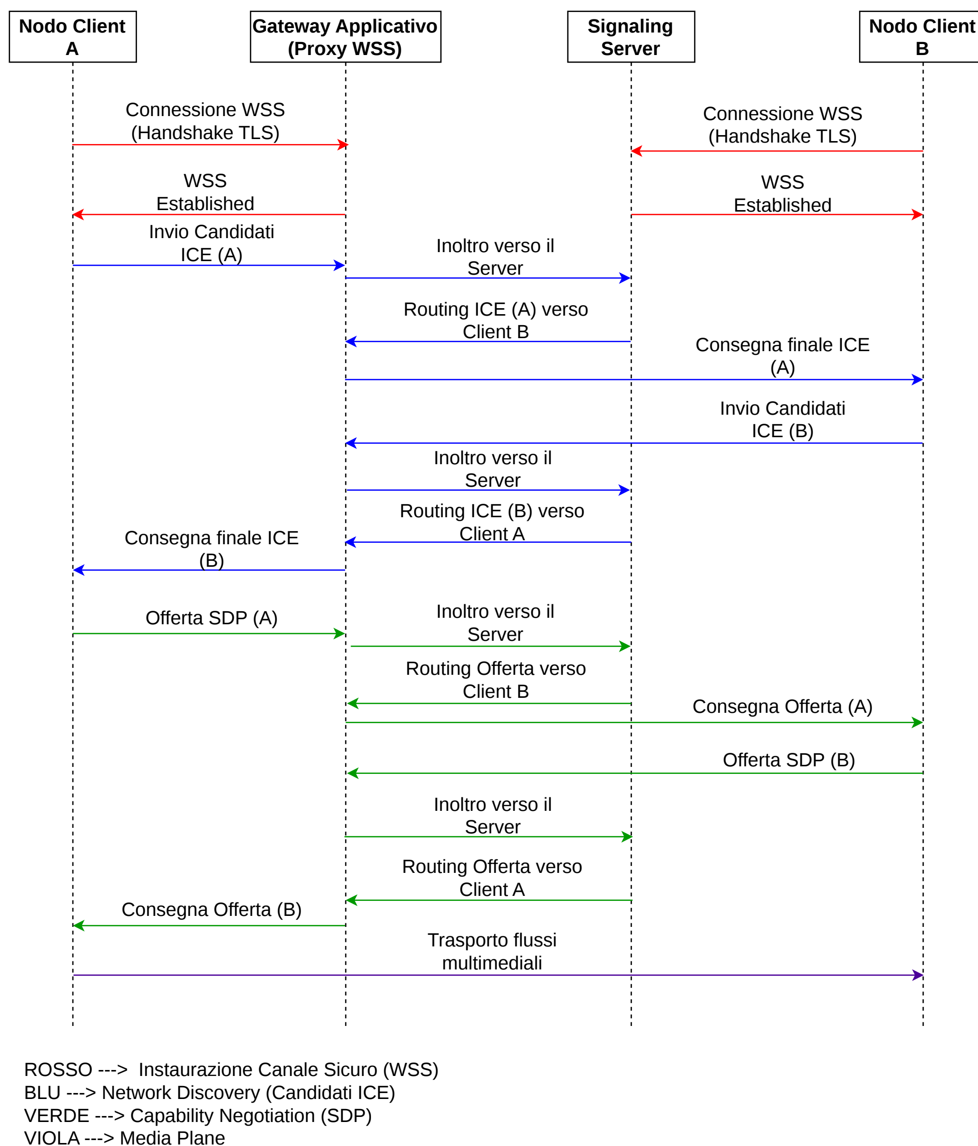

Figura 2.1: Segnalazione WebRTC: instradamento dei candidati ICE e descrittori SDP tramite Gateway (*Reverse Proxy*) e Signaling Server.

2.1.3 Topologie per Comunicazioni di Gruppo

L'architettura P2P diretta risulta inefficiente negli scenari multi-utente: in una chiamata di gruppo, ogni dispositivo dovrebbe inviare una copia del proprio flusso a ciascun partecipante (topologia *Mesh*), saturando rapidamente la banda in *uplink*. Per superare questo limite, si utilizzano server centrali chiamati *Selective Forwarding Unit* (SFU). In questa topologia a stella, l'utente trasmette il proprio flusso multimediale una sola volta al server, che analizza i metadati di rete e lo inoltra selettivamente agli altri partecipanti. L'SFU non decodifica i contenuti, ma si limita a instradare i pacchetti. Questo approccio ottimizza l'uso della rete: il *client* riduce drasticamente i requisiti di banda in *uplink*, delegando all'infrastruttura centrale il compito di distribuire le copie necessarie.

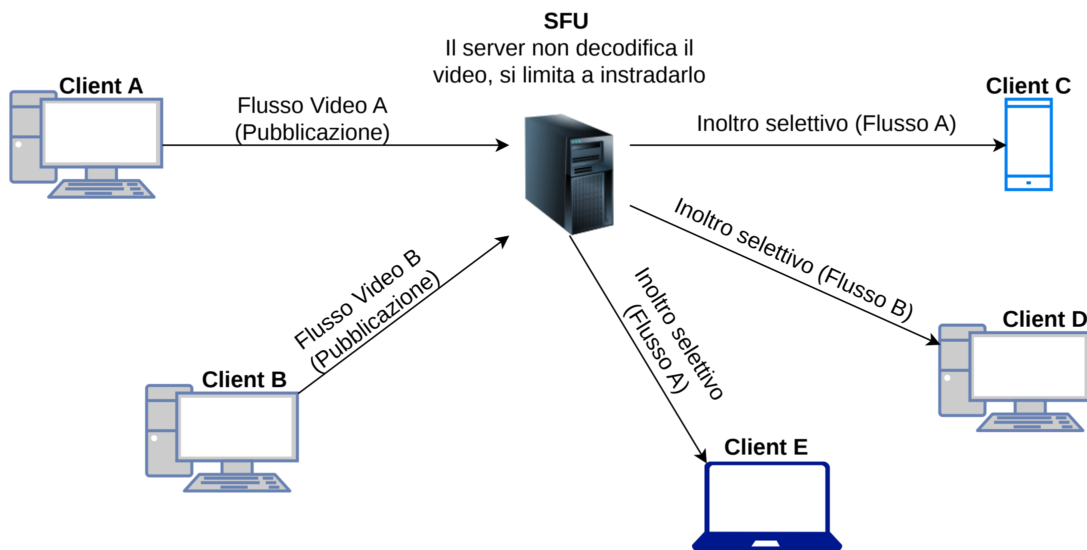

Figura 2.2: Topologia a stella per comunicazioni multi-utente tramite SFU.

2.1.4 Limiti di Sicurezza e Doppia Cifratura

Affinché il server centrale possa instradare i flussi, deve poter leggere gli *header* dei pacchetti in transito. Nel modello di sicurezza standard di WebRTC, i canali di comunicazione sono protetti esclusivamente tramite i protocolli TLS e DTLS. Questa cifratura tutela i dati sulla rete, ma non costituisce una vera sicurezza *End-to-End*

(E2EE), poiché la connessione sicura viene instaurata e terminata tra il singolo *client* e il server.

Di fatto, un SFU moderno **non decifra il contenuto multimediale**, limitandosi a ispezionare gli *header* esterni per un inoltro rapido. Tuttavia, gestendo le connessioni direttamente con i *client*, il server **possiede le chiavi crittografiche** della sessione. Questo approccio, definito *Hop-by-Hop*, presenta una criticità strutturale evidente: qualora l'infrastruttura centrale venisse compromessa, le conversazioni potrebbero essere facilmente decifrate, violando la confidenzialità dei partecipanti.

Affinché la comunicazione rimanga confidenziale anche in presenza di infrastrutture SFU non completamente fidate (*untrusted*), sono necessari due livelli di protezione distinti:

- Una cifratura *Hop-by-Hop* per proteggere i metadati e consentire il corretto instradamento da parte dell'SFU.
- Una cifratura *End-to-End* (E2EE) dedicata esclusivamente alla protezione dei contenuti multimediali.

I primi tentativi di standardizzare questo duplice livello si sono basati sull'estensione SRTP *Double Encryption* (RFC 8723). Pur risolvendo il problema della confidenzialità, questo approccio ha evidenziato gravi limiti strutturali: scarsa efficienza computazionale, elevata complessità implementativa per i server esistenti e un forte accoppiamento con il livello di trasporto RTP. Queste criticità rendono le soluzioni basate su SRTP inadatte e difficilmente scalabili negli scenari moderni.

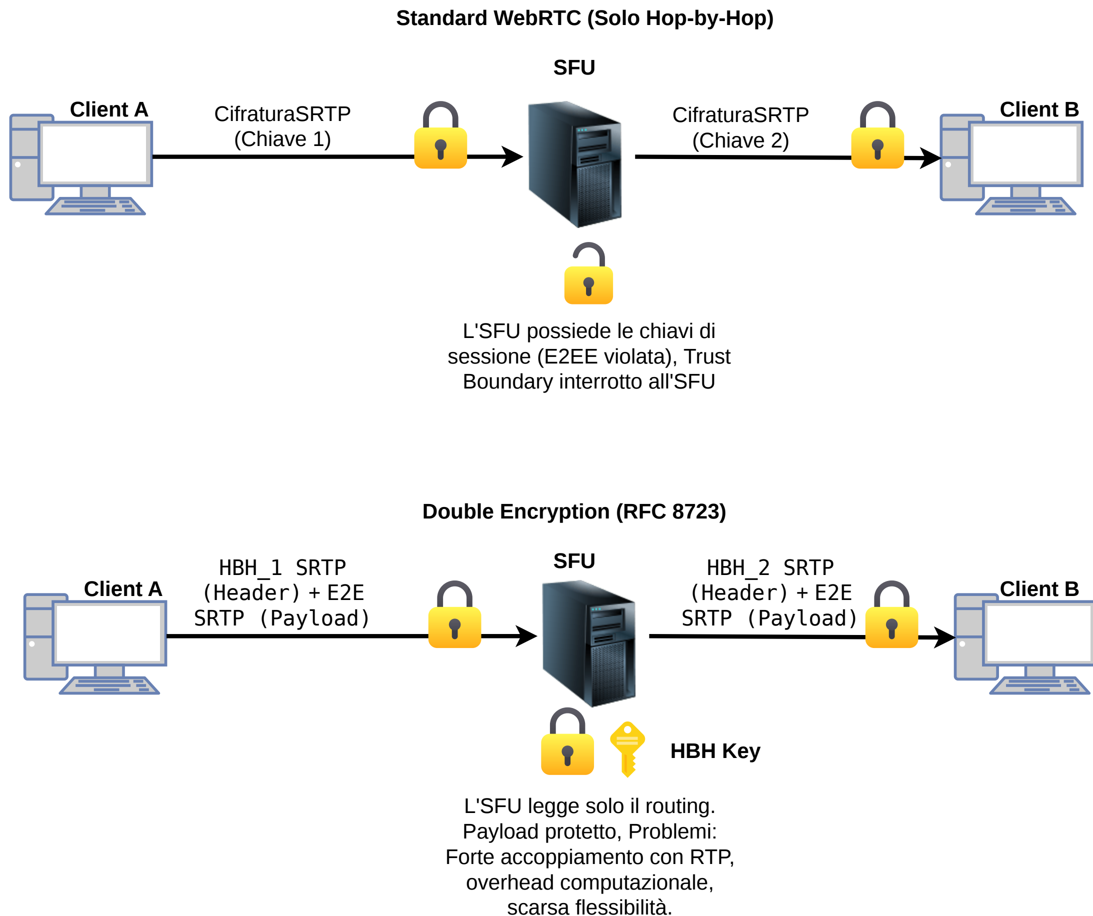

Figura 2.3: Confronto tra l'approccio standard (Hop-by-Hop) e lo schema a Doppia Cifratura.

2.1.5 Meccanismi di Intercettazione dei Flussi Multimediali

Per superare i limiti della doppia cifratura e implementare un'E2EE efficiente, è necessario intervenire sui dati prima che WebRTC li invii al server. Poiché WebRTC gestisce l'acquisizione e la codifica in modo nativo, storicamente le API non permettevano di accedere ai flussi multimediali prima del loro incapsulamento in pacchetti RTP.

Per colmare questa lacuna, il W3C ha introdotto le *WebRTC Encoded Transforms* (o *Insertable Streams*). Tramite le API JavaScript, questa tecnologia fornisce al browser un'interfaccia `TransformStream` che permette di intercettare il singolo fotogramma

subito dopo la codifica. In questo modo è possibile cifrarlo e reinserirlo nel flusso di rete, ponendo le basi architetturali per l'E2EE moderna.

Il ruolo di WebAssembly (WASM):
L'esecuzione di algoritmi crittografici su flussi video in tempo reale richiede massima efficienza e latenza minima. Poiché JavaScript non è ottimizzato per carichi computazionali così intensivi, l'implementazione nei browser si affida a **WebAssembly** (WASM): un formato binario che consente l'esecuzione di codice ad alte prestazioni, scritto in linguaggi come C++ o Rust, direttamente nel Web.

Una volta ottenuto l'accesso ai fotogrammi tramite le *Encoded Transforms*, è necessario definire gli standard per la cifratura e la gestione delle chiavi. Lo stato dell'arte per le comunicazioni di gruppo sicure si basa sull'integrazione di due protocolli distinti definiti dall'IETF. Il protocollo **MLS** (*Messaging Layer Security*) viene impiegato nel *control plane* per gestire in modo efficiente la generazione e la distribuzione delle chiavi tra i partecipanti. Il protocollo **SFrame** (*Secure Frame*) opera invece sul *media plane*, utilizzando le chiavi fornite da MLS per cifrare i singoli fotogrammi multimediali prima della loro trasmissione.

2.2 Protezione del Flusso Multimediale (SFrame)

Lo standard *Secure Frame* (SFrame) si occupa esclusivamente di cifrare il contenuto multimediale a livello dati (*Media Plane*). Il protocollo cifra il fotogramma video mantenendo integri gli *header* esterni: in questo modo, il server centrale può instradare il traffico senza compromettere la confidenzialità *End-to-End*. Rispetto alla precedente *Double Encryption*, SFrame risulta molto più leggero e flessibile poiché esegue solo l'operazione di cifratura, delegando la gestione delle chiavi a sistemi esterni.

Prima di approfondire le specifiche crittografiche, è utile schematizzare le fasi operative del protocollo in quattro passaggi:

1. **Intercettazione del Media:** Il fotogramma generato dall'applicazione viene prelevato in chiaro, tipicamente tramite l'interfaccia *WebRTC Encoded Transforms*.

2. **Derivazione del materiale chiave:** Partendo da una chiave di base (*base_key*, fornita ad esempio da MLS) e dal relativo identificatore (KID), il protocollo impiega una funzione di derivazione (HKDF) per calcolare la chiave di cifratura specifica (*sframe_key*) e un *salt* crittografico.

3. **Cifratura e Autenticazione:** Combinando un contatore (CTR) e il *salt* per generare un *nonce* univoco, il fotogramma viene cifrato con un algoritmo AEAD. Contestualmente viene preparato l'header SFrame, che funge da dati associati in chiaro (AAD).

4. **Trasmissione:** Il *ciphertext* risultante viene affidato allo *stack* WebRTC per l'incapsulamento e la spedizione in rete.

Secondo le specifiche IETF, l'incapsulamento SFrame può avvenire in due modalità: **Per-Packet** (cifrando i singoli pacchetti RTP) o **Per-Frame** (cifrando l'intero fotogramma prima dell'incapsulamento per il trasporto). Quest'ultimo approccio, reso possibile nei browser dai *Transform Streams*, garantisce un'efficienza superiore.

SFrame impiega schemi di **cifratura autenticata con dati associati** (AEAD - *Authenticated Encryption with Associated Data*). Il protocollo non impone un unico algoritmo, ma supporta diverse *cipher suite*: queste includono schemi AEAD nativi (come AES-GCM a 128 o 256 bit) o combinazioni di AES-CTR con HMAC-SHA256 per l'autenticazione. L'output di un'operazione SFrame è strutturato in:

- **SFrame Header (AAD):** Un'intestazione in chiaro che funge da *Additional Authenticated Data* per l'algoritmo AEAD. Contiene il *Key Identifier* (**KID**), che indica al ricevente quale chiave utilizzare, e un Contatore (**CTR**) per prevenire attacchi di *replay*.

- **Payload cifrato e Authentication Tag:** Il fotogramma cifrato seguito da un codice di autenticazione (MAC). Questo tag garantisce l'integrità sia del *payload* che dell'header, assicurando che il pacchetto non sia stato manomesso durante il transito.

2.3 Gestione delle Chiavi di Gruppo (MLS)

Il protocollo *Messaging Layer Security* (MLS) è lo standard IETF per la generazione e distribuzione sicura delle chiavi e opera, in questo contesto, come livello di controllo (*Control Plane*).

A differenza dei protocolli crittografici tradizionali (come il *Signal Protocol*), che si basano su sessioni a coppie obbligando l'utente a scambi separati con ogni membro, MLS organizza i partecipanti in una struttura ad albero (*TreeKEM*). L'efficienza del protocollo deriva dalla possibilità per ogni nodo di ricalcolare la nuova chiave di radice (*root key*) in modo indipendente. Quando un partecipante aggiorna il proprio stato crittografico, non deve trasmettere la nuova chiave singolarmente a ciascun utente: sfruttando la topologia dell'albero, è sufficiente inviare un unico messaggio di aggiornamento (*Commit*). Da questo pacchetto, gli altri membri sono in grado di derivare autonomamente la nuova chiave condivisa. Questo approccio elimina la necessità di sessioni *uno-a-uno*, riducendo il numero di messaggi scambiati e ottimizzando l'uso della rete.

Il concetto di Epoca e Master Secret:
Per mantenere la sincronia tra i partecipanti, MLS suddivide la sessione in stati discreti denominati **Epoche** (*Epochs*). L'Epoca avanza a ogni variazione della composizione del gruppo o dello stato crittografico. All'interno dell'albero, a ogni utente è assegnato un **Indice** univoco (*Leaf Index*) che lo identifica formalmente. A ogni cambio di Epoca, il protocollo calcola un nuovo segreto condiviso definito *Master Secret*. Per ragioni di sicurezza, tale parametro non viene mai impiegato direttamente per la cifratura dei flussi, ma funge da materiale chiave di base: fornendo il *Master Secret* in input a una funzione di derivazione (HKDF), ciascun partecipante genera localmente le sotto-chiavi richieste da SFrame. Questo approccio garantisce la separazione logica e crittografica tra la gestione del gruppo (MLS) e la protezione dei dati (SFrame).

KeyPackage e Welcome Message:
A livello implementativo, il protocollo definisce due strutture dati fondamentali per la gestione dei membri:

- **KeyPackage:** La credenziale pubblica di un utente, contenente le sue chiavi di cifratura e l'identità verificata.

- **Welcome Message:** Un pacchetto cifrato inviato ai nuovi membri, contenente le informazioni necessarie per sincronizzarsi con l'Epoca corrente e derivare il *Master Secret*.

I Servizi Ortogonali: Authentication Service (AS) e Delivery Service (DS)
Il protocollo MLS introduce due servizi logici fondamentali per il suo funzionamento. L'**Authentication Service (AS)** è responsabile della gestione dei certificati e dell'autenticazione dei *client*. Il **Delivery Service (DS)** agisce invece come intermediario di instradamento (*Blind Relay*) e gestore dell'ordine temporale. Il DS riceve i messaggi cifrati (come i *KeyPackage* o i *Welcome*), stabilisce una sequenza cronologica rigorosa per evitare conflitti di stato crittografico tra i partecipanti e li recapita ai destinatari. Poiché il contenuto è protetto da cifratura *End-to-End*, il DS svolge l'instradamento senza poter accedere ai segreti del gruppo. Secondo le specifiche di *framing* (RFC 9420), il server definisce l'ordinamento globale e risolve i conflitti di concorrenza basandosi esclusivamente sui metadati in chiaro presenti nell'involucro esterno del pacchetto (struttura `PrivateMessage`).

Come definito dallo standard, i campi leggibili dal server si limitano all'identificativo del gruppo (`group_id`), all'epoca corrente (`epoch`), alla tipologia di contenuto (`content_type`) e a eventuali dati di autenticazione aggiuntivi (`authenticated_data`). Queste informazioni permettono al DS di serializzare le operazioni senza accedere al *payload*, che rimane protetto all'interno del `ciphertext`.

Capitolo 3

Progettazione

Questo capitolo descrive l'architettura del prototipo di videoconferenza realizzato. Seguendo un approccio *top-down*, viene definita inizialmente la topologia di rete e la gerarchia dei ruoli, identificando le responsabilità di ciascuna entità e i dati necessari al loro funzionamento. Successivamente, l'analisi illustra i protocolli di comunicazione e le interazioni che gestiscono il ciclo di vita di una sessione sicura, dalla fase di inizializzazione crittografica alla trasmissione multimediale.

3.1 Architettura Logica

L'ecosistema implementa i ruoli funzionali previsti dagli standard WebRTC e MLS analizzati nel Capitolo 2. Per fornire una visione architetturale indipendente dalle scelte implementative, la Figura 3.1 illustra come le entità degli standard siano state mappate sui componenti del sistema, assegnando a ciascun nodo responsabilità funzionali delimitate.

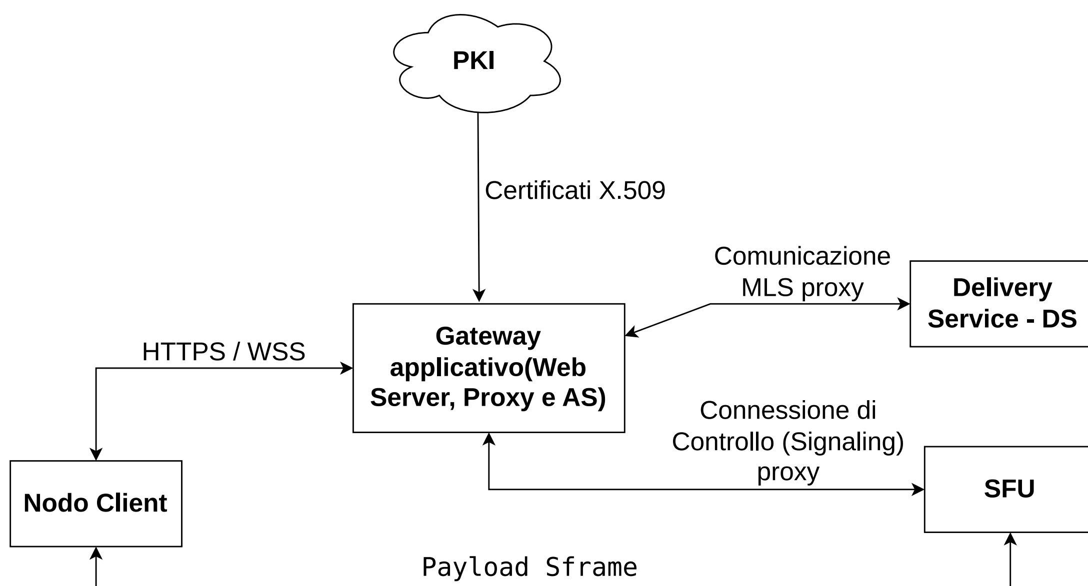

Figura 3.1: Diagramma architetturale: mappatura dei ruoli logici standard sui componenti infrastrutturali e dati statici.

3.1.1 Mappatura dei Componenti e Caratteristiche Funzionali

L'infrastruttura si compone di quattro macro-entità, ciascuna incaricata di ruoli specifici:

- **Nodo Client (Browser Web):**
  - *Ruoli Logici IETF:* **MLS Client** e **SFrame Endpoint**.
  - *Caratteristiche Funzionali:* Rappresenta il software eseguito sul dispositivo dell'utente. È l'unica entità autorizzata ad accedere ai flussi multimediali in chiaro (locali o remoti) ed esegue internamente le operazioni crittografiche E2EE. Mantiene lo stato dell'albero *TreeKEM* per derivare il *Master Secret* e applica la cifratura SFrame (AES-GCM) sui fotogrammi. Garantisce che né i dati multimediali né le chiavi private lascino mai il dispositivo in chiaro.
- **SFU:**
  - *Ruoli Logici IETF:* **Selective Forwarding Unit (SFU)** e **Signaling Server**.

  - *Caratteristiche Funzionali:* Gestisce l'instradamento nel piano di controllo e nel *Media Plane*. Come *Signaling Server*, elabora i messaggi WebRTC (negoziazione SDP/ICE); come SFU, riceve i flussi multimediali per inoltrarli ai destinatari. L'SFU opera esclusivamente a livello di trasporto: analizza gli *header* RTP per lo smistamento, inoltrando il *payload* cifrato da SFrame senza decodificarne o alterarne il contenuto.

- **Delivery Service (DS):**
  - *Ruolo Logico IETF:* **Delivery Service**.
  - *Caratteristiche Funzionali:* Agisce come strato di persistenza e ordinamento per il *Control Plane*. Memorizza il materiale crittografico pubblico (*KeyPackage*) e distribuisce i messaggi di aggiornamento del gruppo (es. *Welcome*). Il DS stabilisce la sequenza temporale degli eventi per evitare conflitti di epoca tra i *client*, operando come un *relay* puro che serializza i pacchetti senza possedere le chiavi per accedervi.

- **Gateway Applicativo e PKI:**
  - *Ruolo Logico IETF:* **Authentication Service (AS)**.
  - *Ruoli Architetturali:* **Web Server** e **Reverse Proxy**.
  - *Caratteristiche Funzionali:* È l'elemento di raccordo dell'architettura, sviluppato *ad hoc* per integrare le logiche standard con l'ambiente applicativo:
    * **Autenticazione (AS per MLS):** Tramite una PKI privata, valida l'identità dell'infrastruttura verso i *client*, fornendo il livello di fiducia richiesto dallo standard MLS.
    * **Orchestrazione e Integrazione:** Distribuisce il *frontend* (HTML/JS) e agisce da *Reverse Proxy* per instradare il traffico verso DS e SFU. Gestisce le terminazioni TLS (HTTPS/WSS) e risolve le *policy* di sicurezza del browser (CORS), fungendo da tramite tra i *client* e i servizi di *backend*.

3.1.2 Dati Statici e Pre-condivisi

Affinché i nodi possano stabilire comunicazioni sicure, l'architettura si affida a un set di dati configurati in fase di *deployment*. Come evidenziato nella Figura 3.1, sono presenti due elementi chiave:

1. **Certificati X.509 (Root CA):** Rappresentano il *trust anchor* del sistema. Memorizzati nel *client* e nel Gateway, sono necessari per instaurare le connessioni TLS (HTTPS e WSS). Senza la validazione di questi certificati, i browser negano la creazione di un *Secure Context*, bloccando l'accesso a periferiche multimediali e alle *WebCrypto API*.

2. **Nomi Host e Configurazione di Rete:** Gli *endpoint* sono definiti staticamente per garantire il corretto instradamento dei flussi. L'architettura impiega un nome host dedicato (es. `sframe.local`) per esporre il Gateway: tale configurazione è indispensabile affinché il browser possa validare la corrispondenza tra l'URL e il certificato X.509. Parallelamente, il sistema utilizza interfacce di rete locali (come l'indirizzo di *loopback* o sottoreti private) e porte dedicate; queste permettono al *Reverse Proxy* del Gateway di indirizzare le richieste verso i processi interni del *Delivery Service* e dell'SFU, mantenendoli isolati dall'accesso pubblico diretto.

3.2 Protocolli di Comunicazione e Logica Ibrida

La creazione e la gestione di una sessione sicura avvengono attraverso una sequenza precisa di passaggi. La Figura 3.2 illustra il flusso temporale di queste comunicazioni tra i Nodi Client e l'infrastruttura di backend, evidenziando il ruolo del Gateway come intermediario (*proxy*) verso il Delivery Service e l'SFU. Inoltre, poiché la gestione completa degli alberi crittografici di MLS all'interno dei browser presenta ancora dei limiti prestazionali, il sistema adotta un **approccio crittografico ibrido**. Questa soluzione combina la logica di organizzazione del gruppo fornita da OpenMLS con gli strumenti crittografici nativi già integrati nel browser (*WebCrypto API*).

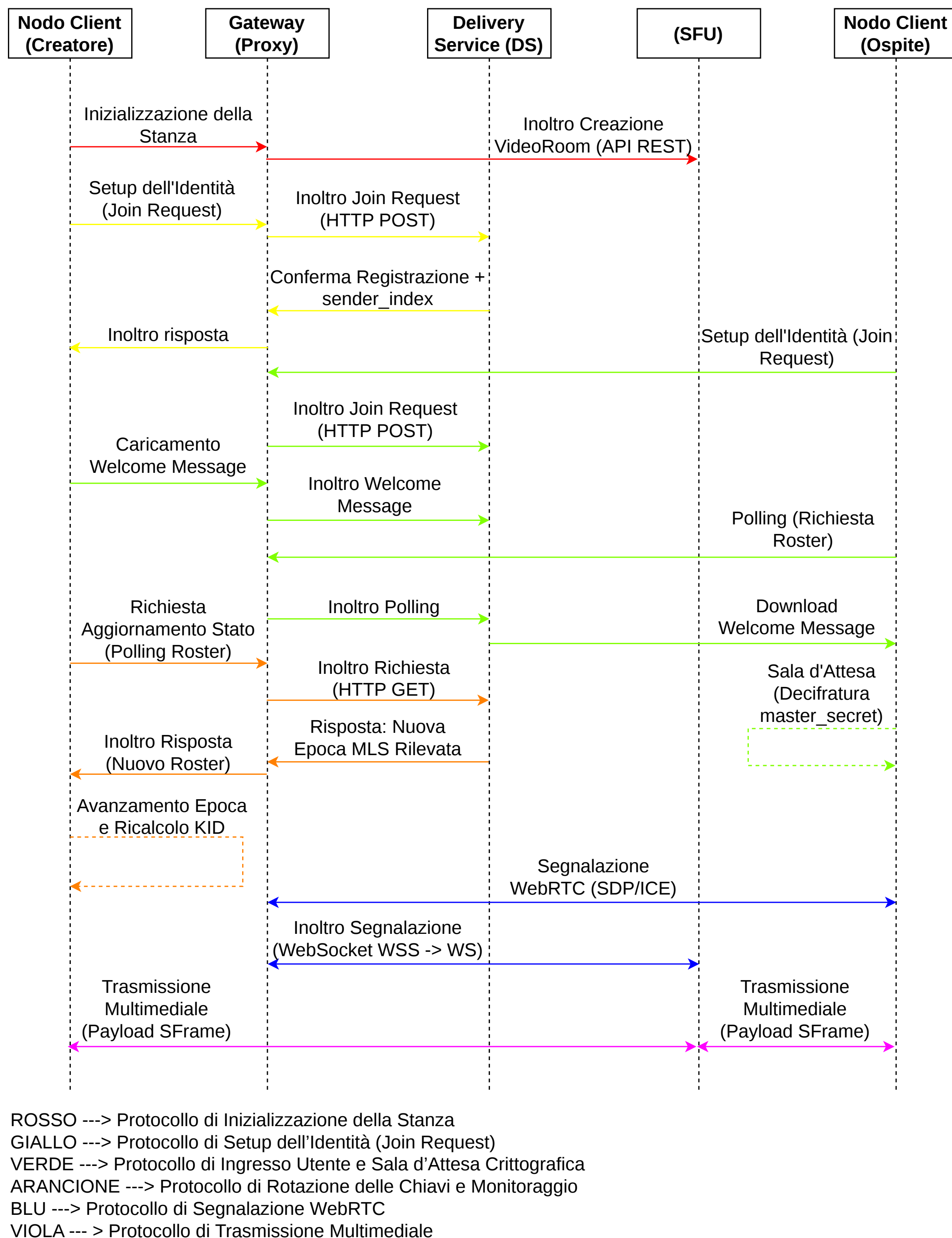

Figura 3.2: Diagramma di sequenza ad alto livello dei protocolli di comunicazione, con mappatura cromatica delle fasi.

3.2.1 Protocollo di Inizializzazione della Stanza

*Questa fase è rappresentata dal colore **rosso** nella Figura 3.2.*

Prima che una conferenza possa ospitare partecipanti, l'utente creatore (*initiator*) esegue il protocollo di creazione. Quando il Nodo Client invia il primo payload al DS, il server rileva che la stanza è vuota e contrassegna l'utente come creatore. A questo punto, il Nodo Client esegue due operazioni:

- **Livello Logico:** Inizializza un nuovo gruppo MLS in locale tramite WebAssembly, diventandone il membro fondatore, e genera crittograficamente un `master_secret` casuale a 256 bit.
- **Livello di Rete:** Contatta il server Janus inviando un comando di creazione stanza al plugin *VideoRoom*. L'SFU alloca le risorse di rete logiche associandole all'ID della stanza precedentemente generato.

3.2.2 Protocollo di Setup dell'Identità (Join Request)

*Questa fase è rappresentata dal colore **giallo** nella Figura 3.2.*

Il protocollo riguarda l'identificazione crittografica dell'utente al momento dell'accesso all'applicativo. Sfruttando il canale TLS sicuro, il Nodo Client genera in locale i propri parametri di sicurezza. Nello specifico, elabora un *KeyPackage* (tramite il modulo nativo *OpenMLS*) e parallelamente genera una coppia di chiavi effimere basate su curve ellittiche (ECDH P-256) tramite le WebCrypto API del browser. Le componenti pubbliche derivanti sia dal modulo MLS che da quello ECDH vengono concatenate in un singolo *payload* JSON codificato in Base64 e trasmesse tramite richiesta HTTP POST al Delivery Service (DS). Il DS memorizza il pacchetto nel *Roster* della stanza, ovvero il registro che tiene traccia di tutti i partecipanti attivi e dei relativi materiali crittografici, assegnando al nuovo utente un indice univoco (`sender_index`) e rendendolo disponibile per futuri inviti.

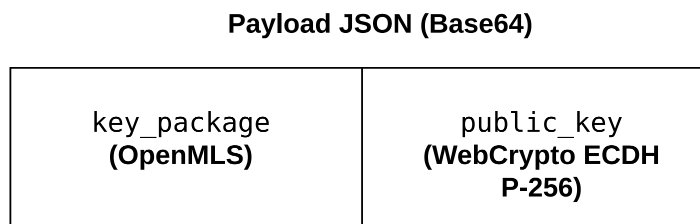

Figura 3.3: Struttura logica del payload ibrido inviato al Delivery Service durante la fase di Join.

3.2.3 Protocollo di Ingresso Utente e Sala d'Attesa Crittografica

Questa fase è rappresentata dal colore **verde** *nella Figura 3.2.*

Questo protocollo governa l'espansione del gruppo e la distribuzione delle chiavi ai nuovi membri. Data la natura *stateless* del Delivery Service, i Nodi Client devono gestire autonomamente l'allineamento dello stato tramite un meccanismo di sincronizzazione periodica (*polling* attraverso la funzione `mlsResync`). Questo processo è essenziale, poiché consente ai client di rilevare nuovi ingressi nel *Roster* (il registro dei partecipanti) o la presenza di nuovi messaggi crittografici, in assenza di un sistema di notifiche *push* dal server.

Quando il **creatore** della sessione (l'utente che ha inizializzato il gruppo) rileva nel *Roster* un nuovo utente in attesa, avvia la procedura di invito. Per trasmettere il segreto in modo sicuro, il creatore preleva la chiave pubblica ECDH del destinatario dal DS e la combina con la propria chiave privata per generare un segreto condiviso (*Shared Secret*) tramite l'algoritmo Diffie-Hellman. Questo segreto derivato viene quindi utilizzato come chiave simmetrica per cifrare il `master_secret` corrente all'interno di un pacchetto AES-GCM.

Questo pacchetto, insieme al *Welcome Message* di OpenMLS, viene caricato sul DS. Grazie al meccanismo di *polling*, il nuovo utente rileva il messaggio, lo scarica e utilizza la propria chiave privata (insieme alla chiave pubblica del creatore) per ricalcolare lo stesso segreto condiviso e decifrare il materiale crittografico. Questa fase definisce una **Sala d'Attesa Crittografica**: il Nodo Client blocca ogni interazione con l'SFU finché non ha completato con successo la decifratura del `master_secret`, garantendo

che nessuna trasmissione multimediale avvenga prima del completo allineamento delle chiavi tra i partecipanti.

3.2.4 Protocollo di Rotazione delle Chiavi e Monitoraggio

*Questa fase è rappresentata dal colore **arancione** nella Figura 3.2.*

L'architettura garantisce la coerenza e la sicurezza del gruppo attraverso il monitoraggio dell'evoluzione temporale della sessione. Ogni modifica strutturale della composizione del gruppo determina l'incremento dell'Epoca globale da parte del Delivery Service. I client operativi rilevano tale variazione in modo asincrono durante i cicli di *polling*: confrontando l'Epoca locale con quella memorizzata nel *Roster*, il client identifica il disallineamento e avvia la procedura di aggiornamento dello stato. Questo processo innesca il ricalcolo del materiale crittografico, derivando dal nuovo *master_secret* i parametri necessari per SFrame, come il *Key Identifier* (KID) e le chiavi di cifratura AES-GCM. Tale meccanismo di rotazione continua assicura il mantenimento delle proprietà di *Forward Secrecy*, invalidando le chiavi delle epoche precedenti a ogni variazione del gruppo e garantendo che un'eventuale compromissione passata non pregiudichi la sicurezza dei flussi futuri.

3.2.5 Protocollo di Segnalazione WebRTC

*Questa fase è rappresentata dal colore **blu** nella Figura 3.2.*

Acquisito lo stato crittografico del gruppo, i Nodi Client avviano il protocollo di *signaling* WebRTC per instaurare il trasporto dati. I client stabiliscono una connessione WebSocket con l'SFU (instradata dal Gateway). In questa fase, i nodi si scambiano i descrittori di sessione in formato SDP e i candidati di rete ICE per determinare il percorso IP ottimale. La procedura culmina con la creazione della *RTCPeerConnection*, il canale bidirezionale dedicato al traffico multimediale opaco.

3.2.6 Protocollo di Trasmissione Multimediale (SFrame)

*Questa fase è rappresentata dal colore **viola** nella Figura 3.2.*

Stabilito il trasporto di rete, si avvia il flusso dati (*Media Plane*). I fotogrammi audio e video catturati dai Nodi Client passano attraverso i *Transform Streams* del browser e vengono intercettati dal modulo WebAssembly. Il codice WASM utilizza la funzione di derivazione HKDF per estrarre le chiavi di trasmissione simmetriche dal `master_secret`. Successivamente calcola il *Key Identifier* (KID) combinando l'Epoca corrente, l'ID della stanza e l'indice del mittente.

I frame in chiaro vengono cifrati con AES-GCM, formando pacchetti compatibili con lo standard SFrame, e trasmessi all'SFU, che li inoltra ai destinatari senza decifrarne né alterarne il contenuto.

Capitolo 4

Implementazione del Sistema

Il presente capitolo descrive il processo di traduzione dell'architettura logica, definita in fase di progettazione, in un sistema software operativo. L'implementazione ha richiesto l'integrazione di tecnologie eterogenee, con l'obiettivo di coniugare la flessibilità tipica dell'ambiente Web con l'efficienza computazionale di un motore crittografico nativo.

La trattazione si focalizza sulle principali scelte implementative, sui pattern di comunicazione tra i diversi layer software e sull'impiego delle API di sistema che rendono possibile l'elaborazione dei flussi in tempo reale. Le sezioni seguenti illustrano la configurazione dell'ambiente di sviluppo, la realizzazione del *core* crittografico in Rust, le tecniche di intercettazione dei flussi multimediali all'interno del browser e la strategia adottata per lo scambio sicuro delle chiavi.

4.1 Stack Tecnologico e Ambiente di Sviluppo

L'implementazione dell'architettura descritta nei capitoli precedenti si basa su uno stack tecnologico e su un ambiente di sviluppo eterogenei. Ogni componente è stato selezionato utilizzando framework e linguaggi specifici, bilanciando requisiti di sicurezza, prestazioni computazionali e flessibilità in fase di prototipazione. La disposizione dei servizi e le tecnologie impiegate sono illustrate nel diagramma di deployment in Figura 4.1.

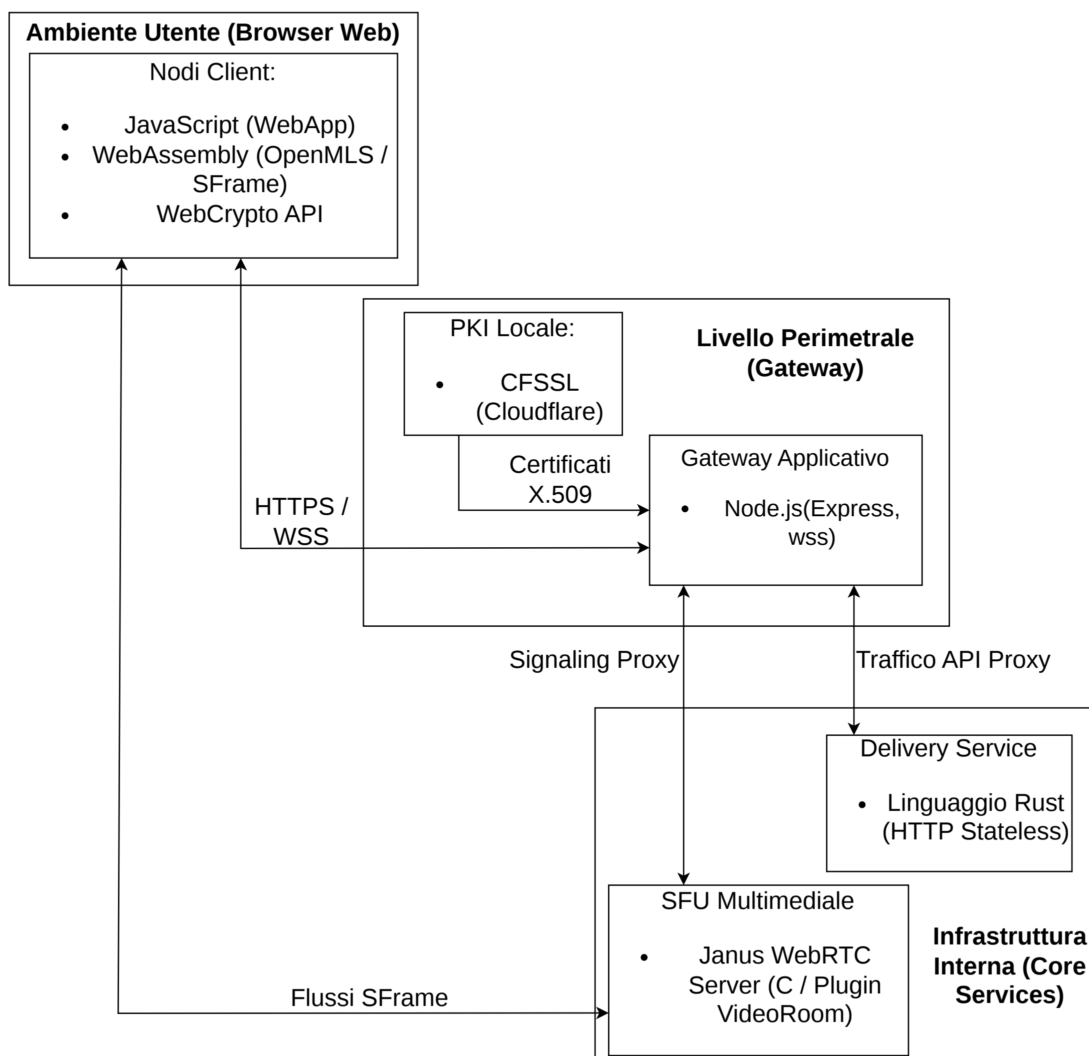

Figura 4.1: Diagramma di deployment a livelli (3-Tier) e stack tecnologico dell'infrastruttura.

Il sistema è stato realizzato mappando le entità architetturali sui seguenti applicativi e strumenti di sviluppo:

- **Gateway Applicativo (Node.js):** Sviluppato utilizzando il runtime *Node.js* (versione 22.11.0) e il framework *Express*. Questo nodo gestisce la distribuzione degli asset statici su HTTPS e agisce da *reverse proxy* per il traffico WebSocket (libreria *ws*). La scelta di Node.js è motivata dalla rapidità di sviluppo e dall'inte-

grazione di middleware per la gestione delle policy CORS e della terminazione TLS.

- **Infrastruttura di Rete (Janus SFU) e PKI:** Il ruolo di SFU è ricoperto da *Janus WebRTC Server*. Janus è stato scelto per la sua natura modulare e per l'architettura basata su plugin (*VideoRoom*), che lo rende uno strumento adatto a contesti di ricerca e sviluppo prototipale, pur garantendo l'efficienza necessaria nell'instradamento dei flussi multi-party. Per quanto riguarda la sicurezza dei canali, è stata realizzata una **PKI privata ad hoc per scopi di test**. Attraverso l'utilizzo di *CFSSL* (toolkit open-source di Cloudflare), è stata istituita una **Root CA locale** per generare i certificati X.509 necessari a validare l'identità del Gateway e abilitare i contesti sicuri (*Secure Context*) richiesti dai browser.

- **Delivery Service (Rust):** Sviluppato in linguaggio *Rust* (compilatore `rustc` 1.90.0), opera come server HTTP indipendente. Rust è stato selezionato per le sue garanzie di *memory safety* e per la gestione efficiente della concorrenza, requisiti fondamentali per un servizio che deve serializzare correttamente le transazioni crittografiche MLS di più client simultanei.

- **Nodi Client e Core Crittografico (JS e WebAssembly):** L'applicativo frontend utilizza **JavaScript** per la gestione della UI e l'orchestrazione WebRTC. Tuttavia, per garantire le prestazioni necessarie alla cifratura *frame-by-frame* senza impattare sulla fluidità del video, il motore crittografico (OpenMLS e SFrame) è stato implementato in **Rust** e compilato in **WebAssembly (WASM)**. L'integrazione tra i due ambienti è gestita tramite `wasm-pack`, che permette al thread JavaScript di invocare le funzioni crittografiche native con un *overhead* minimo.

- **Ambiente di Esecuzione e Test:** I test operativi sono stati condotti in ambiente multipiattaforma (Windows, macOS e Android) utilizzando browser basati su motore *Chromium*. Tale scelta è stata dettata dal supporto maturo di questi browser per le *WebRTC Encoded Transforms*, interfaccia fondamentale per l'intercettazione dei flussi multimediali tramite *Insertable Streams*.

4.2 Il Core Crittografico e l'Integrazione WebAssembly

Il fondamento tecnico del sistema risiede nel modulo crittografico nativo, sviluppato in linguaggio Rust e compilato in WebAssembly (WASM). L'architettura del core è stata progettata definendo due macro-entità indipendenti: `WasmMlsClient`, dedicato alla logica del *Control Plane*, e `WasmPeer`, specializzato nell'elaborazione ad alte prestazioni del *Media Plane*.

L'esposizione di queste strutture verso l'ambiente di esecuzione JavaScript avviene tramite le macro del pacchetto `wasm-bindgen`, che si occupa della traduzione dei tipi di dato (ad esempio mappando i vettori di byte di Rust in strutture dati compatibili con le API Web). L'interazione tra queste entità è illustrata nel diagramma architetturale in Figura 4.2.

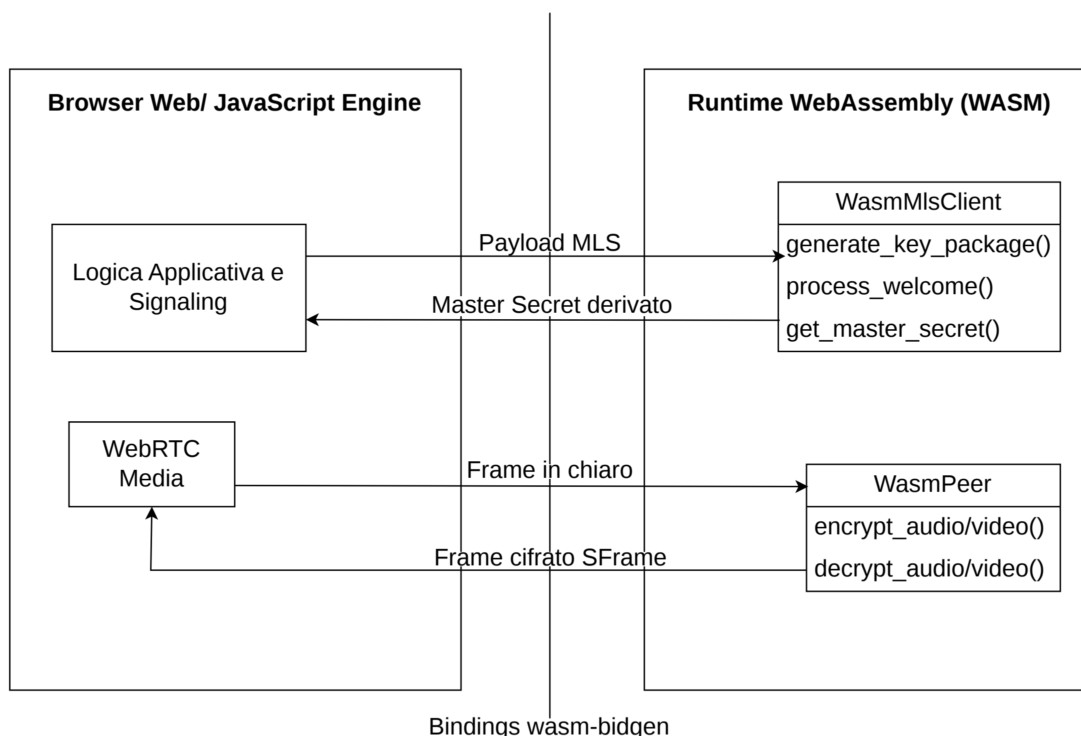

Figura 4.2: Diagramma delle entità logiche del modulo WebAssembly e relative interfacce di comunicazione con l'ambiente JavaScript.

4.2.1 Gestore delle Identità (WasmMlsClient)

La classe `WasmMlsClient` rappresenta l'interfaccia logica per la gestione dello stato crittografico del singolo nodo. **In fase di inizializzazione**, il componente configura il contesto crittografico (*Crypto Provider*) e genera le coppie di chiavi necessarie alla **validazione dell'identità** del client all'interno del protocollo. I metodi esposti alla WebApp orchestrano l'evoluzione dello stato del gruppo:

- Il metodo `generate_key_package()` produce il *KeyPackage* dell'utente. Tale struttura definisce l'identità crittografica del nodo, consentendone la registrazione all'interno del gruppo. In particolare, il pacchetto rende pubblici i dati necessari affinché il creatore della sessione possa inserire il nuovo partecipante nell'albero dei segreti (*Ratchet Tree*) e far avanzare l'Epoca (*Epoch*). In questa implementazione, il protocollo OpenMLS è utilizzato per gestire la struttura del gruppo, mentre il segreto di sessione (*Master Secret*) è gestito esternamente tramite le WebCrypto API.

- Il metodo `process_welcome()` interviene durante l'accesso alla stanza: riceve il *payload* MLS, ne verifica la validità e inizializza lo stato locale del gruppo. Ciò consente al nuovo nodo di sincronizzarsi con l'Epoca corrente e con la gerarchia dell'albero.

- Il metodo `get_master_secret()` estrae la chiave di sessione derivata dall'accordo di chiave di gruppo (*Group Key Agreement*) e la rende disponibile al livello applicativo. Questo passaggio fornisce il segreto necessario alla cifratura dei flussi multimediali SFrame.

4.2.2 Gestore dell'Elaborazione Media (WasmPeer)

Il componente `WasmPeer` implementa la logica di elaborazione in tempo reale dei flussi multimediali. Al fine di garantire l'isolamento dei dati e prevenire conflitti nei contatori crittografici, la classe mantiene stati distinti per i canali di trasmissione e ricezione di audio e video.

Durante la fase di inizializzazione vengono acquisiti il *Master Secret* e gli identificativi di chiave (*Key ID*); questi ultimi sono gestiti come interi a 64 bit (`u64`) per garantire la coerenza dei tipi di dato ed evitare errori di *overflow* durante l'interazione tra l'ambiente Rust e l'interfaccia JavaScript. I metodi operativi principali includono:

- `encrypt_video()` e `encrypt_audio()`: ricevono il buffer di dati in chiaro dai codificatori del browser, applicano la cifratura autenticata AES-GCM tramite la libreria nativa e restituiscono il frame cifrato con l'aggiunta dell'intestazione SFrame.
- `decrypt_video()` e `decrypt_audio()`: eseguono l'operazione di decifratura in ricezione, verificando l'integrità del messaggio tramite il tag di autenticazione prima di inoltrare il frame decifrato allo stack WebRTC.

L'architettura integra inoltre una funzionalità di telemetria basata sulla struttura `SframeHeaderDebug`. Durante ogni operazione crittografica, il modulo estrae i metadati rilevanti (quali il contatore dei frame, il KID attivo e la dimensione del payload). Tali informazioni sono rese accessibili all'applicazione per finalità di monitoraggio e debug in tempo reale, senza incidere sulle prestazioni di elaborazione dei flussi.

4.3 Architettura del Client Web e Organizzazione dei Moduli

L'architettura del Client Web è stata progettata per integrare il modulo crittografico nativo all'interno di un ecosistema modulare e sicuro. Il progetto (denominato *sframe_project*) segue il paradigma della *Separation of Concerns*, isolando l'interfaccia utente, la logica di segnalazione e le procedure crittografiche in domini software indipendenti.

A livello di macro-struttura, l'organizzazione dei moduli si articola in tre componenti principali:

1. **Logica Applicativa e Interfaccia (Client):** Rappresenta il punto di interazione con l'utente, implementato tramite moduli JavaScript. Questo livello gestisce lo

stato della sessione e l'interazione con le API del browser, coordinando in modo asincrono l'orchestrazione WebRTC e il flusso di dati con il core crittografico WASM.

2. **Core Crittografico (WebAssembly):** Costituito dai binari compilati ospitati nella directory `pkg`. Questo modulo include il file `sframe_core_bg.wasm` e i relativi *bindings*, che permettono al browser di eseguire operazioni MLS e SFrame con prestazioni elevate.

3. **Infrastruttura di Supporto e Orchestrazione (Gateway):** Il sistema è supportato dal modulo Node.js `secure_server.js`, che centralizza l'accesso alle risorse di backend. Questo componente svolge tre funzioni critiche:

    - **Web Serving:** Espone l'applicazione su protocollo HTTPS, distribuendo gli asset statici (HTML, JS, WASM) in un contesto sicuro, requisito obbligatorio per l'utilizzo delle WebCrypto API.
    - **Orchestrazione Sessioni:** Implementa una logica di backend dedicata tramite l'endpoint `POST /api/new-room`. Questa funzione automatizza la comunicazione con Janus (via HTTP REST) per la creazione delle sessioni e delle stanze multimediali (*VideoRoom*), semplificando il flusso operativo del client.
    - **Reverse Proxying:** Gestisce l'instradamento delle connessioni WebSocket (*WSS to WS*) verso Janus e delle chiamate REST verso il *Delivery Service* MLS. Tale configurazione permette al browser di interagire con un unico punto di ingresso sicuro, mantenendo il server Gateway estraneo alla conoscenza delle chiavi crittografiche di sessione.

4.3.1 Logica Applicativa del Client

La logica del client si basa su un'architettura modulare, in cui ogni script gestisce aspetti specifici dell'applicazione:

- **Livello di Interfaccia (`ui.js` e `output.js`):** Gestiscono l'interazione con il *Document Object Model* (DOM) e il sistema di logging. Aggiornano l'interfaccia

grafica (notifiche e griglie video) in risposta agli eventi asincroni, mantenendosi separati dalla logica di controllo.

- **Orchestratore Principale (`appRoom.js`):** È il punto di ingresso dell'applicazione. Gestisce la connessione WebSocket verso il gateway, coordina la **segnalazione WebRTC** (scambio di SDP e candidati ICE) e fa da raccordo tra gli eventi di rete e l'interfaccia utente.
- **Gestione del Control Plane (`mls_sframe_session.js`):** Implementa la logica crittografica di gruppo. Amministra l'istanza `WasmMlsClient`, monitora l'avanzamento delle Epoche e usa le WebCrypto API per derivare e scambiare i segreti di sessione.
- **Gestione del Media Plane (`sframe_layer.js`):** Funge da interfaccia tra JavaScript e il modulo `WasmPeer`. Esegue le conversioni di tipo necessarie per far comunicare l'ambiente JS con l'architettura a 64 bit di Rust (ad esempio, il *casting* dei Key ID in formato `BigInt`).

Questa suddivisione isola le operazioni computazionalmente più onerose e la gestione dello stato all'interno dei moduli di basso livello. Al controller `appRoom.js` resta il compito di sincronizzare gli eventi e instradare i flussi verso il modulo di cifratura SFrame, garantendo un codice lineare e facilmente manutenibile.

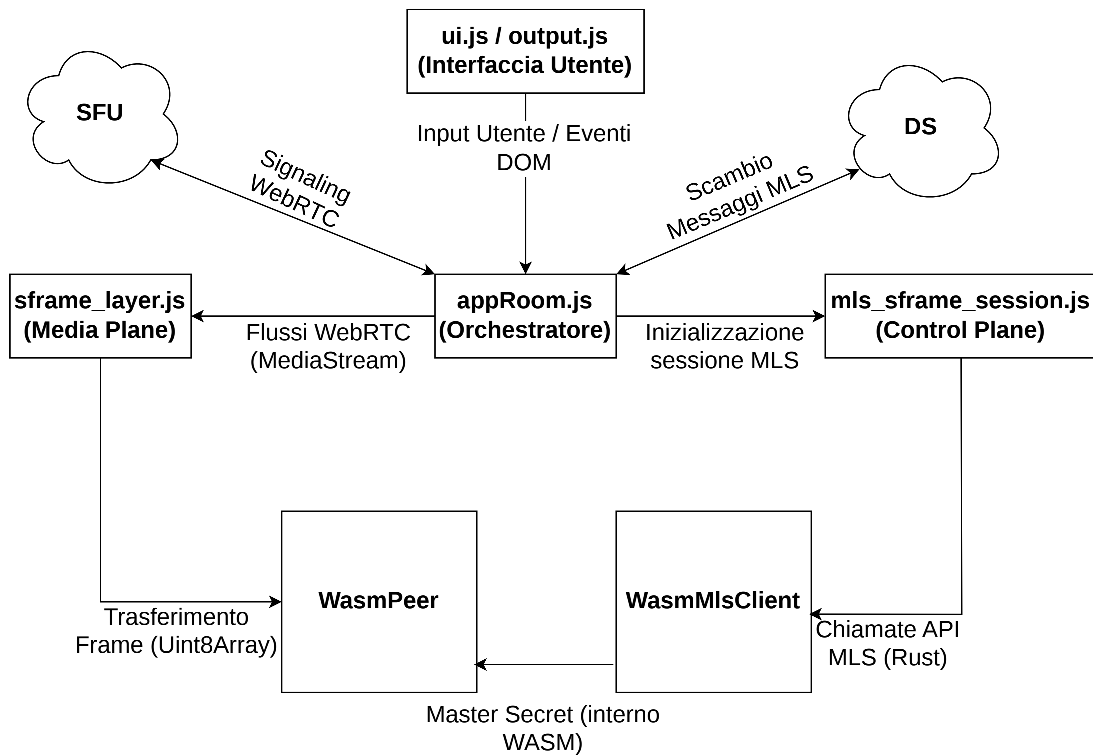

Figura 4.3: Architettura modulare del client web e gerarchia dei flussi. L'orchestratore (*appRoom.js*) coordina la segnalazione verso l'SFU e il Delivery Service (DS), delegando l'elaborazione del piano di controllo (MLS) e del piano multimediale (SFrame) ai moduli core in WebAssembly.

4.4 Architettura Ibrida e Delivery Service E2EE

Come delineato nelle sezioni precedenti, l'implementazione del *Control Plane* si basa su una strategia ibrida. Questa scelta progettuale è finalizzata a massimizzare l'affidabilità nello scambio delle chiavi, mitigando i limiti **strutturali** legati alla persistenza dello stato in WebAssembly quando interfacciato con l'ambiente asincrono di JavaScript.

In questa configurazione, mentre la validazione delle identità e l'avanzamento temporale del gruppo (le *Epoche*) restano sotto il controllo del protocollo MLS, la distribuzione del *Master Secret* è gestita tramite un livello parallelo basato sulle **WebCrypto API**. Il

coordinamento di tale scambio, come illustrato nell'architettura di sistema, è orchestrato dal server dedicato sviluppato in Rust.

4.4.1 Il Delivery Service

Il *Delivery Service* è un server HTTP asincrono che opera parallelamente all'SFU Janus. Il suo ruolo è fungere da intermediario per l'instradamento dei messaggi crittografici di controllo: il server riceve e smista esclusivamente pacchetti cifrati, senza alcuna visibilità sulle chiavi di sessione o sui contenuti in chiaro. L'impiego del formato Base64 è limitato alla codifica dei dati binari per garantirne la compatibilità con il trasporto su protocollo HTTP, mentre l'opacità del carico utile (*payload*) è garantita dagli algoritmi crittografici del livello *End-to-End*.

Il server gestisce lo stato di ciascuna stanza in memoria (`GroupState`), tenendo traccia della lista dei partecipanti (*Roster*) e dell'Epoca corrente. Le interazioni principali avvengono tramite tre *endpoint* REST:

- `POST /mls/join`: Un nuovo client deposita le proprie credenziali pubbliche. Il server assegna un indice univoco (`sender_index`) e informa il client se è il creatore della stanza (`is_creator`).
- `POST /mls/welcome`: Utilizzato dal creatore della stanza per recapitare un invito cifrato a un nuovo utente. La ricezione di questo messaggio fa avanzare in modo incrementale e centralizzato l'Epoca del gruppo (`gs.epoch += 1`).
- `GET /mls/roster`: Un *endpoint* interrogato ciclicamente dai client (tramite *polling* in background) per rilevare l'ingresso di nuovi partecipanti, l'avanzamento dell'Epoca o la ricezione di nuovi messaggi *Welcome*.

4.4.2 Il Flusso Crittografico nel Client Web

Il modulo JavaScript `mls_sframe_session.js` rappresenta il fulcro dell'orchestrazione logica del protocollo ibrido e coordina sia il core WASM sia le WebCrypto API, seguendo un flusso rigoroso:

1. **Creazione del Payload Combinato:** Quando un utente accede alla stanza, invoca il modulo `WasmMlsClient` per generare il proprio *KeyPackage* MLS. Contestualmente genera una coppia di chiavi effimere per l'algoritmo *Elliptic Curve Diffie-Hellman (ECDH)*. Questi due elementi vengono aggregati in un singolo oggetto JSON e serializzati in Base64 prima di essere inviati al Delivery Service.

2. **Gestione da parte del Creatore:** L'utente che inizializza la stanza genera localmente un *Master Secret* casuale di 32 byte. Quando rileva un nuovo utente nel *Roster*, utilizza la chiave pubblica ECDH del nuovo arrivato per derivare un segreto condiviso e impiega l'algoritmo AES-GCM per cifrare il *Master Secret*. Il risultato viene caricato sul server come *Welcome* combinato.

3. **Apertura dell'Invito:** Il nuovo nodo, una volta scaricato il proprio *Welcome*, utilizza la propria chiave privata ECDH per estrarre il *Master Secret* in chiaro e allinearsi all'Epoca corrente.

L'intera sequenza temporale di negoziazione, comprensiva della fase di attesa (polling), dell'aggiornamento dell'Epoca e della successiva sincronizzazione dello stato locale da parte del creatore, è illustrata nel diagramma di sequenza in Figura 4.4.

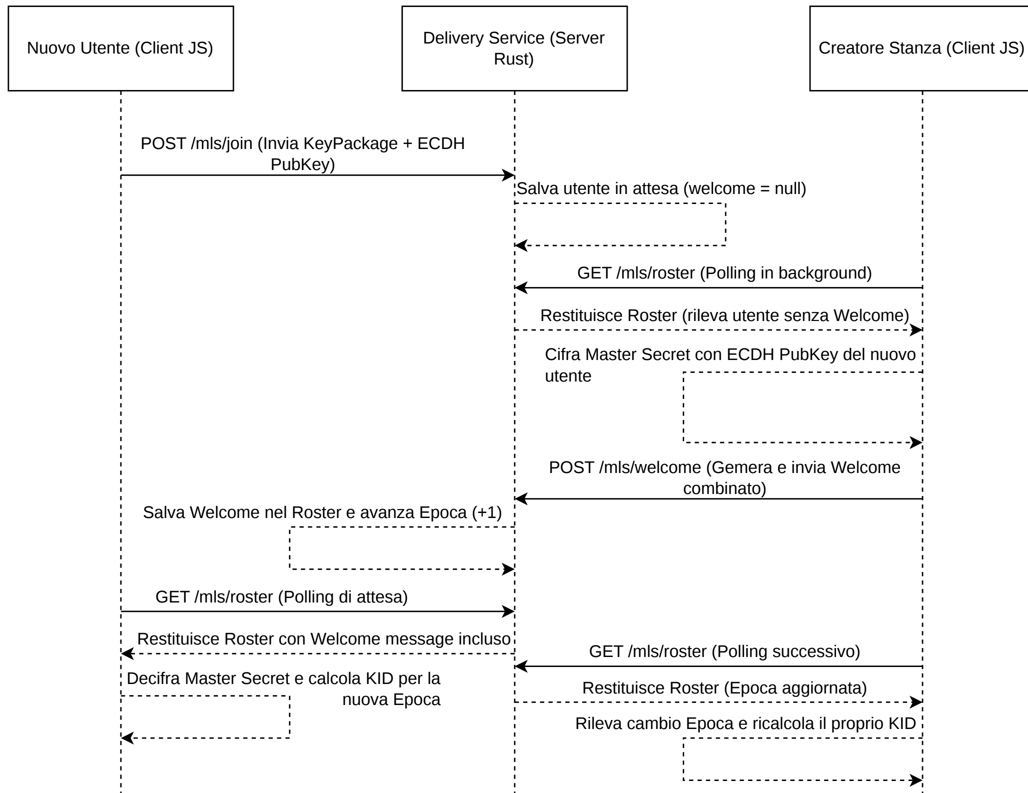

Figura 4.4: Diagramma di sequenza delle operazioni crittografiche e di sincronizzazione dell'Epoca tra i peer e il Delivery Service.

4.4.3 Derivazione delle Chiavi di Trasporto e Calcolo del KID

Una volta distribuito il *Master Secret*, la derivazione delle chiavi crittografiche simmetriche finali è demandata al modulo `hkdf.js`. Utilizzando le primitive WebCrypto, il sistema applica la funzione **HKDF-HMAC-SHA256** per derivare chiavi di invio (*TxKey*) e ricezione (*RxKey*) distinte per ogni partecipante, concatenando il segreto base con un'etichetta specifica (*label*) che include l'indice del mittente.

Infine, al fine di inizializzare i *Transform Streams* dedicati all'elaborazione multimediale, i client calcolano in modo autonomo e deterministico il *Key ID* (KID) da inserire negli header SFrame. Per garantire l'assenza di collisioni crittografiche in un ambiente multi-stanza senza un coordinatore centrale delle chiavi, l'algoritmo di calcolo mappa linearmente i parametri di sessione in un singolo intero a 64 bit:

$$\text{KID} = (\text{Epoca} \times 10^9) + (\text{RoomId} \times 10^4) + (\text{SenderIndex} \times 10)$$

Essendo inserito in chiaro nell'header SFrame di ciascun pacchetto, questo identificativo consente ai ricevitori di recuperare immediatamente la chiave di sessione corretta, validando il tag di autenticazione del frame audio o video prima del suo inserimento nella pipeline WebRTC, come verrà analizzato nella sezione successiva.

4.5 Intercettazione dei Flussi Multimediali (Transform Streams)

Una volta completato lo scambio delle chiavi e stabilito il materiale crittografico di sessione, la cifratura *End-to-End* sui flussi audio e video all'interno del browser è garantita tramite le **WebRTC Insertable Streams API**. Questa tecnologia permette di intervenire nella pipeline multimediale standard, estraendo i singoli frame codificati (dal codec VP8 o Opus) appena prima che vengano affidati al livello di trasporto di rete, o immediatamente dopo la loro ricezione.

Nel sistema implementato, l'intercettazione avviene all'interno del modulo principale (`appRoom.js`), dove vengono istanziati dei `TransformStream` personalizzati sia per le connessioni in uscita (Sender) che per quelle in ingresso (Receiver).

4.5.1 Pipeline di Trasmissione (Sender)

Il metodo `createEncodedStreams()` viene invocato sull'istanza `RTCRtpSender` per attivare l'intercettazione dei pacchetti al momento dell'avvio della trasmissione. All'interno del ciclo di trasformazione, ogni frammento di dati codificati (*EncodedChunk*) viene convertito in un `Uint8Array` e passato al modulo WebAssembly tramite i metodi `encrypt_video()` o `encrypt_audio()` dell'entità `WasmPeer`.

Il modulo Rust applica la cifratura autenticata AES-GCM e restituisce il frame protetto in formato SFrame, che sostituisce il *payload* originale prima di essere reimmesso nella coda di invio tramite la funzione `controller.enqueue()`. Contestualmente, il ciclo JavaScript interroga l'ambiente WASM mediante l'esportazione `sframe_last_tx_header()`, estraendo i metadati crittografici (KID, contatori e tag di autenticazione) per finalità di telemetria e monitoraggio, senza impattare sulle prestazioni.

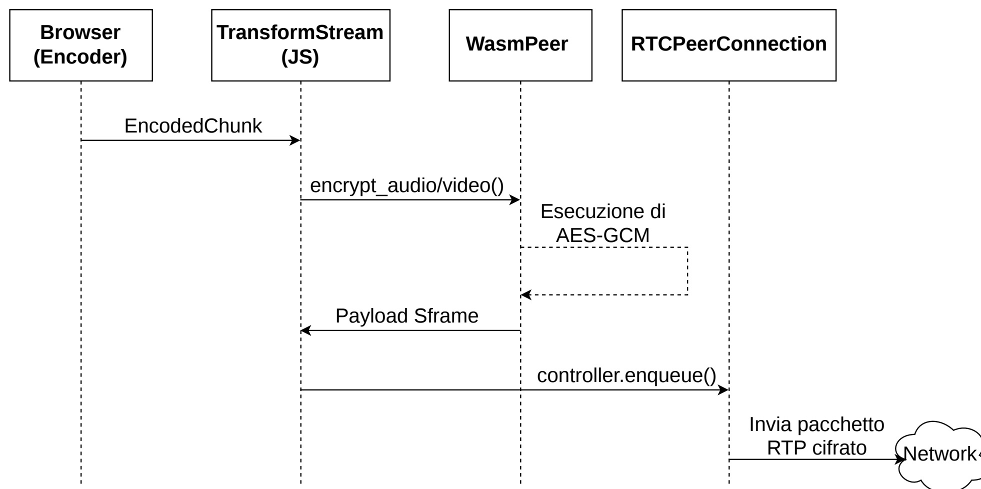

Figura 4.5: Diagramma di sequenza dell'intercettazione e cifratura dei frame. Il processo illustra l'interazione sincrona tra il ciclo di trasformazione JavaScript e il core crittografico in WebAssembly.

4.5.2 Pipeline di Ricezione (Receiver)

In modo analogo, le tracce remote ricevute tramite il server Janus vengono elaborate dai componenti `RTCRtpReceiver`. I frame cifrati sono passati alle funzioni `decrypt_video()` e `decrypt_audio()` del modulo `WasmPeer`, configurato con il materiale segreto derivato per lo specifico mittente.

Particolare attenzione è stata dedicata alla **resilienza durante le transizioni di Epoca**. Poiché WebRTC utilizza principalmente il protocollo UDP per il trasporto dei media, non è garantito l'ordine di arrivo dei pacchetti rispetto ai messaggi di controllo MLS (trasmessi su protocollo affidabile TCP). Di conseguenza, è possibile la ricezione di frame cifrati con una chiave non ancora disponibile nel *Control Plane*.

Per gestire tale disallineamento, il `TransformStream` implementa una logica di gestione delle eccezioni: qualora il modulo Rust segnali l'assenza della chiave necessaria (`DecryptionKeyError`), l'applicazione intercetta l'errore e scarta il frame. Questa strategia evita l'instabilità della pipeline WebRTC, limitando l'effetto visivo a un breve congelamento della riproduzione fino alla corretta sincronizzazione delle chiavi di sessione.

Capitolo 5

Implementazione e discussione sperimentale

Lo sviluppo di questo sistema crittografico *End-to-End*, **in cui l'onere computazionale della cifratura è demandato esclusivamente ai client**, ha richiesto di confrontarsi direttamente con i limiti pratici delle tecnologie web attuali. Poiché SFrame e MLS sono standard ancora emergenti, durante l'implementazione del progetto è emersa una netta differenza tra quanto previsto dalle specifiche teoriche e quanto le librerie software attualmente disponibili consentono realmente di fare.

La scarsa maturità di questi strumenti ha imposto di rivedere l'architettura iniziale per aggirare gli ostacoli tecnici riscontrati. Nella pratica, infatti, l'integrazione tra i protocolli non è stata immediata come descritto sulla carta. Ad esempio, far comunicare la libreria che gestisce i gruppi MLS con le API del browser che applicano la cifratura SFrame ha richiesto lo sviluppo di soluzioni su misura per l'adattamento e il passaggio delle chiavi, superando vincoli e incompatibilità non previsti a livello protocollare. Le sezioni seguenti analizzano questi ostacoli e le relative soluzioni ingegneristiche adottate.

5.1 Limiti di astrazione in WebRTC: l'incompatibilità con LiveKit

Nella fase iniziale di progettazione, la scelta del server SFU era ricaduta su *LiveKit*. Sebbene questa piattaforma metta a disposizione moderni *Software Development Kit* (SDK) progettati per semplificare e velocizzare l'integrazione lato client, oltre a elevate prestazioni e un'infrastruttura nativamente orientata al cloud, il suo elevato livello di astrazione si è rivelato strutturalmente incompatibile con i requisiti della cifratura *End-to-End* basata su SFrame.

Per intercettare e cifrare i flussi multimediali nel browser, è strettamente necessario l'uso delle *WebRTC Insertable Streams API*. Le specifiche del W3C impongono un vincolo temporale rigido: la funzione `createEncodedStreams()` deve essere invocata sull'oggetto `RTCRtpSender` **prima** che la negoziazione dei parametri di rete (l'handshake SDP) venga conclusa.

Gli SDK di LiveKit, tuttavia, sono progettati per automatizzare e astrarre completamente la gestione della `RTCPeerConnection`. Non appena una traccia multimediale viene affidata alla libreria, questa avvia immediatamente la negoziazione SDP in background in modo trasparente per lo sviluppatore. Di conseguenza, nel momento in cui la logica applicativa tenta di accedere al `RTCRtpSender` sottostante per agganciare il modulo crittografico WASM, la connessione risulta già stabilita. Questo disallineamento temporale generava sistematicamente l'eccezione nativa del browser:

```
InvalidStateError: Too late to create encoded streams
```

rendendo architetturalmente impossibile l'iniezione del `TransformStream`.

Questa criticità ha evidenziato un requisito architetturale fondamentale: l'implementazione di protocolli E2EE *custom* richiede un controllo granulare e a basso livello sul ciclo di vita di WebRTC. Tale requisito ha motivato la transizione definitiva verso *Janus WebRTC Server*, il quale, operando a un livello di astrazione nettamente inferiore, lascia al client la piena responsabilità temporale sulla creazione delle offerte SDP e sull'iniezione dei *Transform Streams*.

5.2 Persistenza dello stato in WebAssembly

L'integrazione del protocollo MLS all'interno del browser, effettuata compilando la libreria nativa `openmls` in WebAssembly, ha fatto emergere criticità significative legate alla gestione della memoria *stateful* in scenari altamente asincroni. Nello specifico, per *stateful* si intende la necessità del modulo crittografico di mantenere un contesto interno persistente (contenente chiavi, identificatori e l'albero dei segreti) tra invocazioni successive separate da tempi di attesa variabili e imprevedibili.

Il protocollo MLS richiede che, quando un client desidera unirsi a un gruppo, generi un pacchetto pubblico (`KeyPackage`) e contestualmente salvi la relativa chiave privata nel proprio *Keystore* locale. Il problema strutturale si presentava nel tempo di attesa tra l'invio del `KeyPackage` al server e la successiva ricezione dell'invito cifrato (`Welcome`).

Questa problematica nasce dal contrasto architetturale tra due paradigmi di esecuzione profondamente diversi. Da un lato, WebAssembly opera su un modello di **memoria lineare**: un unico blocco contiguo di byte (gestito internamente dal codice Rust) in cui variabili e strutture dati devono rimanere allocate fisicamente. Dall'altro lato, JavaScript si basa su un *Event Loop* asincrono a singolo thread, in cui le operazioni di rete (come l'attesa di una risposta dal Delivery Service) sono modellate tramite **Promise**. A livello concettuale, una *Promise* è un'astrazione che incapsula lo stato e il risultato futuro di un'operazione asincrona; durante l'attesa della sua risoluzione, JavaScript non blocca il thread principale, ma sospende l'esecuzione di quel blocco specifico, restituendo il controllo all'*Event Loop* per elaborare nel frattempo altri eventi.

In questa fase di sospensione, se l'ambiente JavaScript non mantiene un riferimento forte e globale all'istanza dell'oggetto crittografico allocato nel modulo WASM, entra in gioco il **Garbage Collector** (GC). Quest'ultimo è il componente del motore JavaScript incaricato di deallocare automaticamente la memoria degli oggetti non più referenziati. Il GC, rilevando l'oggetto wrapper del Keystore come temporaneamente inattivo durante la lunga attesa di rete, ne liberava lo spazio.

Di conseguenza, al momento della ricezione del messaggio di `Welcome`, quando il modulo WASM tentava di interrogare il proprio *Keystore* per recuperare la chiave privata, il puntatore alla memoria lineare risultava invalido. L'esito era un errore

bloccante nell'esecuzione del protocollo crittografico, identificato dall'eccezione nativa `NoMatchingKeyPackage`.

Questa dinamica ha dimostrato che l'integrazione senza riadattamenti architetturali di librerie crittografiche originariamente concepite per ambienti di esecuzione sequenziali nativi risulta fragile all'interno dell'ecosistema WASM quando esposta al modello a eventi del Web.

Per superare tale limitazione, senza rinunciare ai vantaggi di MLS per la gestione delle identità e delle Epoche, è stata progettata l'architettura crittografica ibrida descritta nel Capitolo 4. Delegando lo scambio del *Master Secret* di SFrame alle **WebCrypto API** native del browser, si è risolta la criticità legata alla volatilità della memoria WASM. Le WebCrypto API, essendo implementate in C++ direttamente nel motore del browser, gestiscono la persistenza sicura delle chiavi, risultando immuni ai cicli del Garbage Collector.

Tuttavia, questo approccio impone un preciso compromesso ingegneristico (*trade-off*). Se da un lato garantisce la realizzazione di un *Control Plane* affidabile, dall'altro l'impiego aggiuntivo dell'algoritmo asimmetrico ECDH introduce un inevitabile *overhead* computazionale. Questo rallenta la fase di ingresso dei nuovi utenti nella *VideoRoom*, generando un picco di latenza durante l'inizializzazione della sessione. Tale ritardo, tuttavia, è circoscritto alla sola fase di *setup* e si mantiene entro tempi di attesa accettabili per l'utente, senza compromettere la fluidità del successivo scambio multimediale e garantendo al contempo la sicurezza nel processo di derivazione delle chiavi.

5.3 Valutazione Qualitativa e Quantitativa: L'Impatto dell'Architettura E2EE

I test pratici sul prototipo hanno permesso di valutare l'impatto reale dell'architettura proposta sui due flussi di comunicazione. Durante le sessioni di test, è emerso fin da subito un impatto percettibile sulla latenza applicativa: l'introduzione della logica crittografica asimmetrica (gestita da OpenMLS e dalle API WebCrypto) ha generato

ritardi misurabili e temporanei fenomeni di *freezing* del flusso video in concomitanza dell'ingresso di nuovi utenti nella stanza.

5.3.1 Metodologia di Misurazione: Il Control Plane

Per indagare l'origine di queste latenze e superare le sole osservazioni qualitative, si è deciso di concentrare l'analisi quantitativa sulle performance del *Control Plane*. L'obiettivo è stato misurare il tempo impiegato dal sistema per integrare un nuovo nodo all'interno della *VideoRoom*, aggiornare lo stato crittografico del gruppo (avanzamento dell'Epoca) e processare i messaggi di *Welcome*.

Per ottenere misurazioni accurate, sono state incapsulate le chiamate asincrone della libreria OpenMLS e le operazioni crittografiche ECDH associate con `performance.now()`, un'API ad alta risoluzione messa a disposizione dai browser. Questa scelta metodologica è stata dettata dalla necessità di isolare il tempo di elaborazione crittografica sul *main thread* del browser, escludendo le latenze di rete. Il carico di queste operazioni, infatti, dipende esclusivamente dal numero di partecipanti presenti nella stanza, risultando indipendente dalle dinamiche dei flussi multimediali.

5.3.2 Valutazione Qualitativa: Il Media Plane

Per quanto riguarda il *Media Plane*, la valutazione è stata condotta attraverso un approccio prevalentemente empirico. Trattandosi dell'algoritmo simmetrico AES-GCM (eseguito in WebAssembly dal modulo SFrame), ottimizzato per l'elaborazione di elevati volumi di dati e tipicamente supportato da accelerazione hardware, i test non hanno evidenziato colli di bottiglia prestazionali direttamente imputabili alla cifratura in tempo reale.

Sottoponendo il sistema a un carico di rete e computazionale progressivamente crescente, partendo da risoluzioni base (320x240 a 20 fps) fino a raggiungere l'Alta Definizione (1280x720 a 30 fps), la fluidità del video non ha mostrato degradi evidenti riconducibili all'applicazione dell'algoritmo crittografico sui pacchetti multimediali.

È importante precisare che estrarre metriche prestazionali assolute e isolate per il flusso multimediale all'interno di WebRTC è un'operazione estremamente complessa. Come

evidenziato dalla letteratura tecnica sul tema, tra cui la documentazione *WebRTC for the Curious*, il *debugging* e la profilazione di questa tecnologia rappresentano una sfida significativa. L'architettura WebRTC è composta da numerosi componenti asincroni in continuo mutamento (*encoder*, *packetizer*, *jitter buffer*, algoritmi di stima della banda) che interagiscono tra loro e possono introdurre latenza in modo indipendente. Isolare il contributo computazionale dei *Transform Streams* di SFrame all'interno di questa pipeline, senza ottenere dati fuorvianti, richiederebbe strumenti di *tracing* nativi del motore C++ del browser. Per questo motivo, l'indagine si è concentrata sulle metriche empiricamente più rilevanti del *Control Plane*.

5.3.3 Analisi dei Tempi di OpenMLS

I dati raccolti sul *Control Plane* confermano le osservazioni qualitative iniziali. La complessa interazione asimmetrica necessaria per garantire l'avanzamento sicuro dell'Epoca introduce un costo computazionale che si riflette direttamente sui tempi di esecuzione, traducendosi nella latenza applicativa descritta.

Il grafico e la tabella seguenti illustrano i tempi necessari per completare le operazioni crittografiche di inserimento di un nuovo utente, misurati in millisecondi in funzione del numero totale di partecipanti già attivi all'interno della stanza.

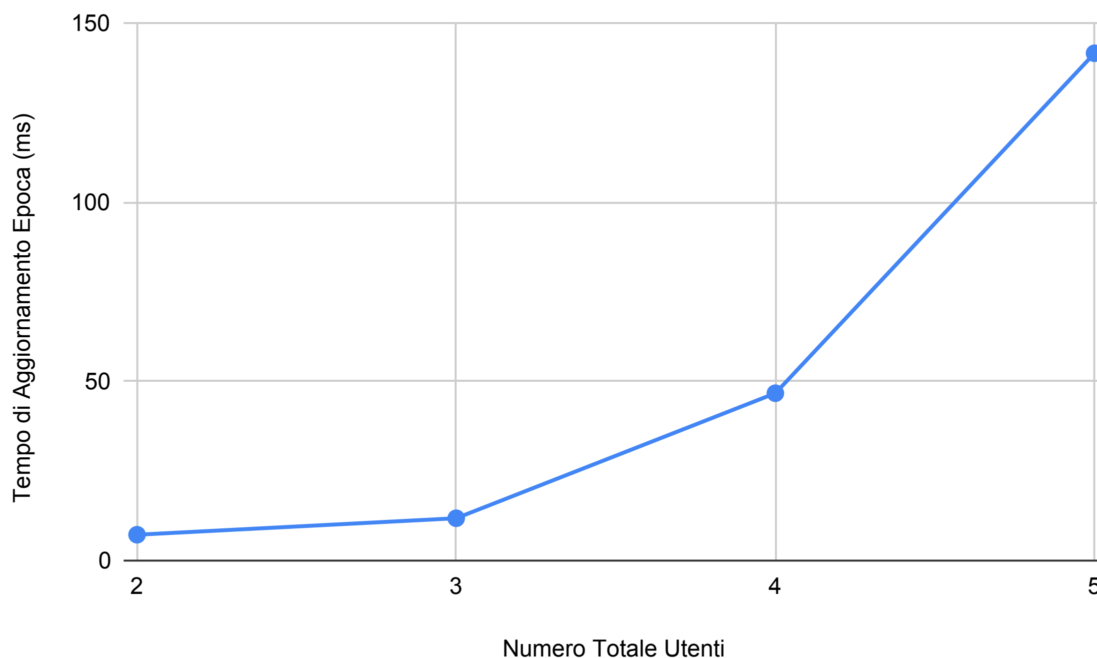

Figura 5.1: Andamento del tempo di aggiornamento dell'Epoca MLS al variare del numero di partecipanti nella stanza.

5.4 Maturità degli Standard e Scelte Implementative: SFrame vs MLS

Il progetto ha dimostrato che *Secure Frame* (SFrame) rappresenta uno standard solido e maturo per garantire la confidenzialità *End-to-End* dei flussi multimediali all'interno di architetture mediate da SFU. A differenza dello schema a "doppia cifratura" SRTP, caratterizzato da elevata complessità e da un forte accoppiamento col trasporto di rete, SFrame soddisfa i requisiti architetturali di disaccoppiamento: separa nettamente la crittografia dal livello di instradamento. Questo permette a un server centrale come Janus di smistare i pacchetti leggendo solo le intestazioni, garantendo la totale opacità dell'infrastruttura rispetto ai contenuti multimediali.

Durante la fase di sviluppo, la selezione degli strumenti crittografici si è orientata fin da subito verso soluzioni compatibili con l'ambiente Web. Per questo motivo, la

scelta è ricaduta in prima istanza sul progetto `sframe-rs` (nel *fork* di *TobTheRock*), un'implementazione interamente scritta in Rust. Altre alternative esaminate, come l'implementazione di riferimento fornita da Cisco (`cisco/sframe`), sono state scartate a priori: essendo sviluppate in C++ e fortemente dipendenti da librerie crittografiche native (come OpenSSL) e da complessi *toolchain* di compilazione (CMake), risultavano inadatte a una portabilità agevole all'interno del browser. Al contrario, la libreria `sframe-rs` non solo garantisce una rigorosa aderenza allo standard IETF (RFC 9605), ma offre anche supporto nativo per la compilazione in WebAssembly. Sfruttando il *backend* `rust-crypto`, è stato possibile eseguire algoritmi complessi come AES-GCM e derivazioni HKDF direttamente nel *client* web, mantenendo prestazioni elevate.

Al contrario, l'implementazione del protocollo *Messaging Layer Security* (MLS) in ambiente browser risulta ancora in una fase di consolidamento. Per la gestione del piano di controllo, la scelta si è orientata su `OpenMLS`, un'implementazione rigorosa della RFC 9420 sviluppata anch'essa in Rust e attivamente manutenuta. Rispetto ad altre alternative, OpenMLS si è distinta per l'architettura sicura e per il supporto esplicito alla compilazione *target* WebAssembly.

L'utilizzo congiunto di `sframe-rs` e `OpenMLS` ha rappresentato una scelta strategica fin dalle prime fasi di design: ha permesso di consolidare l'intero motore crittografico *client-side* attorno a un unico ecosistema (Rust/WASM), ottimizzando i processi di compilazione e sfruttando le garanzie di *memory safety* del linguaggio Rust.

Tuttavia, come analizzato in precedenza, il porting in WebAssembly di librerie native così complesse evidenzia ancora limitazioni critiche nella gestione della memoria e della persistenza dello stato all'interno di ecosistemi asincroni guidati da eventi (JavaScript). Sebbene il modello teorico di MLS (basato su *Ratchet Tree*) sia altamente efficiente per la gestione logica dei gruppi, la sua applicazione pratica in ambito Web impone oggi l'adozione di architetture ibride di compromesso. Ciò evidenzia come, a fronte della validità teorica dei protocolli, l'ecosistema di strumenti per il supporto nativo e fluido di standard crittografici avanzati nel browser necessiti ancora di un'ulteriore fase di maturazione.

Capitolo 6

Conclusioni e Sviluppi Futuri

Il presente lavoro di tesi si è posto l'obiettivo di affrontare una delle criticità più rilevanti nelle moderne architetture di videoconferenza di gruppo: garantire una reale confidenzialità *End-to-End* (E2EE) senza rinunciare alla scalabilità offerta dai server centralizzati. Al termine delle attività di analisi, sviluppo e sperimentazione, è possibile formulare una valutazione complessiva dell'architettura proposta, tracciando un bilancio tra i risultati teorici attesi e i riscontri pratici ottenuti.

Il risultato principale di questo percorso è stato la realizzazione di un prototipo funzionante che dimostra la fattibilità di un'infrastruttura crittografica basata sull'integrazione di due standard IETF emergenti: *Secure Frame* (SFrame) e *Messaging Layer Security* (MLS). L'architettura sviluppata conferma la validità del modello che prevede una separazione tra il piano di controllo e il piano multimediale. Demandando la gestione delle chiavi di gruppo a MLS e la cifratura dei pacchetti video a SFrame, il sistema riesce a svincolare la sicurezza dei dati dal livello di trasporto. Questo approccio affronta le criticità di privacy delle tradizionali topologie SFU (*Selective Forwarding Unit*), garantendo che l'infrastruttura centrale, in questo caso gestita da Janus, operi come un semplice nodo di instradamento, privo del materiale crittografico necessario per accedere in chiaro ai flussi audio e video degli utenti.

Nonostante la solidità dell'architettura, lo sviluppo pratico ha richiesto un confronto diretto con lo stato dell'arte delle tecnologie web. I test condotti hanno evidenziato come l'implementazione di algoritmi crittografici avanzati interamente *client-side* sia possibile,

ma non priva di criticità. Se da un lato SFrame si è rivelato uno strumento solido e performante per l'elaborazione continua dei flussi multimediali, dall'altro l'integrazione di MLS all'interno del browser ha fatto emergere le attuali limitazioni delle librerie compilate in WebAssembly. La gestione dello stato crittografico asincrono e la complessa interoperabilità con JavaScript hanno generato latenze durante le fasi di aggiornamento del gruppo. Questi risultati mostrano che, sebbene la direzione tracciata dai nuovi standard rappresenti il futuro delle comunicazioni sicure, l'ecosistema degli strumenti di sviluppo necessita ancora di una fase di maturazione.

Sulla base di queste considerazioni, il prototipo sviluppato non rappresenta un punto di arrivo, ma un solido *Proof of Concept* (PoC) che si presta a diverse evoluzioni. In prospettiva, è possibile delineare tre principali direzioni di sviluppo futuro.

Il primo passo riguarda l'integrazione del sistema con un *Identity Provider* esterno. Attualmente, l'identità dei nodi all'interno del gruppo MLS è gestita in modo semplificato. Per rafforzare l'infrastruttura, il meccanismo di generazione delle credenziali dovrà essere delegato a sistemi standardizzati di *Identity and Access Management*, come ad esempio Keycloak, sfruttando protocolli consolidati quali OAuth2 e OpenID Connect. Questa evoluzione permetterebbe di certificare l'identità crittografica dei partecipanti in modo affidabile, prevenendo attacchi di impersonificazione e garantendo la coerenza dell'intera catena di autenticazione.

Un secondo ambito di intervento riguarda l'estensione delle garanzie di sicurezza nel tempo, abilitando il *Ratcheting* nativo previsto dalle specifiche SFrame. Sebbene il prototipo attuale garantisca la segretezza dei dati, l'attivazione di una rotazione continua e automatica delle chiavi di sessione è necessaria per implementare la cosiddetta *Post-Compromise Security*. Questo meccanismo consentirebbe di preservare la sicurezza del sistema anche nel caso in cui un attaccante comprometta una singola chiave, garantendo sia la *Forward Secrecy* per le sessioni passate sia la *Post-Compromise Security* per quelle future, grazie al rinnovo costante e unidirezionale del materiale crittografico.

Infine, per superare la natura prototipale e garantire l'affidabilità richiesta da applicazioni reali, l'infrastruttura di backend dovrà essere riprogettata in ottica di scalabilità orizzontale. L'adozione di paradigmi basati su microservizi e tecnologie di containerizzazione permetterebbe una chiara separazione delle diverse componenti del sistema: il

Gateway Applicativo Node.js, il server WebRTC Janus e il Delivery Service scritto in Rust. Tale architettura distribuita consentirebbe di gestire le risorse in modo elastico, ottimizzando il carico di rete e mantenendo prestazioni stabili all'aumentare del numero di utenti simultaneamente connessi.

In conclusione, questo lavoro di tesi presenta un'architettura validata e una chiara roadmap di sviluppo, mostrando come sia possibile coniugare la fluidità di una video-conferenza di gruppo con solide garanzie di sicurezza e privacy offerte dalla crittografia moderna.

Bibliografia

[1] Amazon Web Services (AWS Labs). mls-rs: Rust implementation of the MLS protocol. GitHub repository. https://github.com/awslabs/mls-rs.

[2] B. Beurdouche et al. The Messaging Layer Security (MLS) Architecture. RFC 9750, Internet Engineering Task Force (IETF), 2025. https://www.rfc-editor.org/rfc/rfc9750.html.

[3] Cisco Systems. mlspp: C++ implementation of the Messaging Layer Security (MLS) protocol. GitHub repository. https://github.com/cisco/mlspp.

[4] Cisco Systems. sframe: C++ implementation of Secure Frame (SFrame). GitHub repository. https://github.com/cisco/sframe.

[5] Danilo Bellocchio. sframe_project: Prototipo di videoconferenza cifrata End-to-End con SFrame e MLS. GitHub repository, 2025. https://github.com/danisystem/sframe_project.

[6] H. Alvestrand. Overview: Real-Time Protocols for Browser-Based Applications. RFC 8825, Internet Engineering Task Force (IETF), 2021. https://datatracker.ietf.org/doc/html/rfc8825.

[7] OpenMLS Project. OpenMLS: A Rust implementation of the Messaging Layer Security (MLS) protocol. GitHub repository. https://github.com/openmls/openmls.

[8] R. Barnes et al. The Messaging Layer Security (MLS) Protocol. RFC 9420, Internet Engineering Task Force (IETF), 2023. https://datatracker.ietf.org/doc/rfc9420/.

[9] S. DuBois et al. WebRTC for the Curious: Go beyond the APIs. Open Source Book, 2020. https://webrtcforthecurious.com/.

[10] S. Nandakumar et al. Media over QUIC Transport. Internet-Draft, Internet Engineering Task Force (IETF), 2026. https://datatracker.ietf.org/doc/draft-ietf-moq-transport/.

[11] The Rust Project Developers. The Rust Programming Language Documentation. Official Rust Documentation. https://doc.rust-lang.org/stable/.

[12] TobTheRock. sframe-rs: Rust implementation of Secure Frame (SFrame). GitHub repository. https://github.com/TobTheRock/sframe-rs.

[13] Y. Omara et al. Secure Frame (SFrame). RFC 9605, Internet Engineering Task Force (IETF), 2024. https://www.rfc-editor.org/rfc/rfc9605.html.