FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TP: Room RTC

[95.08] Taller de programación
Segundo Cuatrimestre 2025

Integrantes

Alumno	Padrón
Villa Jimenez, Alexander	95428
Pinargote Velásquez, Tom Harry	111689
Olalla Togra, Nervo Falconery	111731

Índice

Introducción

El presente trabajo constituye la entrega final del proyecto "Room RTC". El objetivo principal ha sido desarrollar una solución de videoconferencias completa y funcional implementando desde cero el stack tecnológico WebRTC en lenguaje Rust.

A diferencia de las etapas preliminares, esta versión final evoluciona de una simple prueba de concepto local a un sistema distribuido robusto. La solución integra ahora un Servidor de Señalización para el descubrimiento de pares, gestión de usuarios y autenticación, así como

una capa de seguridad estricta (TLS, DTLS y SRTP) que garantiza la privacidad e integridad de las comunicaciones en entornos reales.

Alcance

El alcance del proyecto final abarca la implementación total de los siguientes componentes:

- **Signaling Server (Servidor Central):** Desarrollo de un servidor TCP concurrente con protocolo propio para el registro de usuarios, manejo de presencia (Lobby) y enrutamiento de mensajes SDP.
- **Seguridad Integral:** Implementación de TLS para señalización, DTLS para el handshake P2P y una implementación manual de SRTP (AES-128 + HMAC-SHA1) para el cifrado de video.
- **Stack WebRTC Completo:** Módulos de SDP (negociación), ICE y RTP/RTCP (transporte y control).
- **Procesamiento de Medios:** Captura, codificación (OpenH264) y renderizado de video, sumado a un algoritmo de **Control de Congestión** que ajusta la calidad según el estado de la red.
- **Aplicación de Cliente:** Interfaz gráfica (GUI) que integra todos los módulos, permitiendo login, llamadas y visualización de video seguro.

Análisis del problema

En una primera instancia para encarar el proyecto, comenzamos por diagramar la arquitectura completa a alto nivel, para de esa manera entender mejor el problema. Luego, decidimos enfocarnos solo en la parte del cliente(ya que es lo necesario para la entrega intermedia), de esa manera fuimos separando cada unos de los componentes del stack de RTC, en módulos. Por ejemplo, identificamos módulos para SDP, ICE, otro módulo para relacionarlos, otro para el codec, etc.

Metodología de trabajo y herramientas

Dividimos el proyecto en sprints de trabajo de 1 semana y usando “Github Project” fuimos elaborando tarjetas/tareas de desarrollo, para cada feature a implementar. Por lo general, nos dividimos las tareas, por ejemplo, 2 personas se encargaban de implementar SDP y las otras 2 de ICE.

Arquitectura del sistema

Arquitectura General (cliente-servidor)

Para la versión final del proyecto, la arquitectura evoluciona de un modelo puramente P2P local a un modelo híbrido asistido por un servidor central. El sistema se compone ahora de tres nodos principales: dos clientes (peers) y un **Servidor de Señalización (Signaling Server)**.

La comunicación se estructura en dos planos diferenciados:

- **Plano de Señalización (Control):** Se establece una conexión persistente **TCP** protegida por **TLS** entre cada cliente y el servidor. A través de este canal viajan los mensajes de registro, autenticación, cambios de estado (Lobby) y el intercambio de mensajes SDP (Oferta/Respuesta).
- **Plano de Medios (Datos):** Una vez negociada la sesión, los clientes establecen una conexión **P2P UDP** directa. Esta conexión implementa **ICE** para hallar candidatos, **DTLS** para el intercambio seguro de claves y **SRTP** para la transmisión encriptada de audio y video.

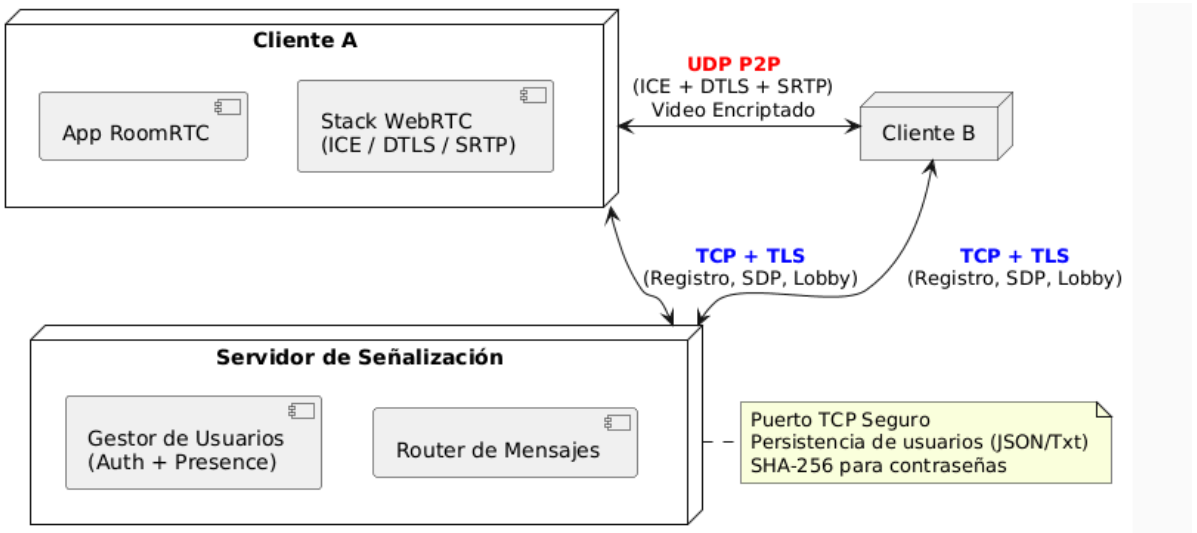

Figura 1.1 - Arquitectura del sistema general

Arquitectura del Cliente

La arquitectura interna del cliente se ha robustecido para integrar la comunicación con el servidor y las capas de seguridad. Se mantiene el patrón de diseño basado en un **Engine** central que orquesta los eventos, pero se integran nuevos módulos:

- **Signaling Client:** Módulo encargado de mantener la conexión TCP/TLS con el servidor. Gestiona el protocolo de mensajería (Login, Oferta, Respuesta) y notifica al Engine sobre eventos de red externos.
- **Capa de Seguridad (Security Layer):** Se incorporan dos componentes críticos en

el transporte:
    - **DTLS (Datagram Transport Layer Security):** Utilizado sobre ICE para el intercambio seguro de claves de encriptación.
    - **SRTP (Secure Real-time Transport Protocol):** Intercepta los paquetes RTP antes de ser enviados (o al recibirlos) para encriptar/desencriptar el payload de video utilizando **AES-128** y **HMAC-SHA1**.
- **Control de Congestión:** Un nuevo módulo que monitorea las estadísticas de envío (RTT, pérdida de paquetes) para ajustar el bitrate del codificador y evitar saturar la red.

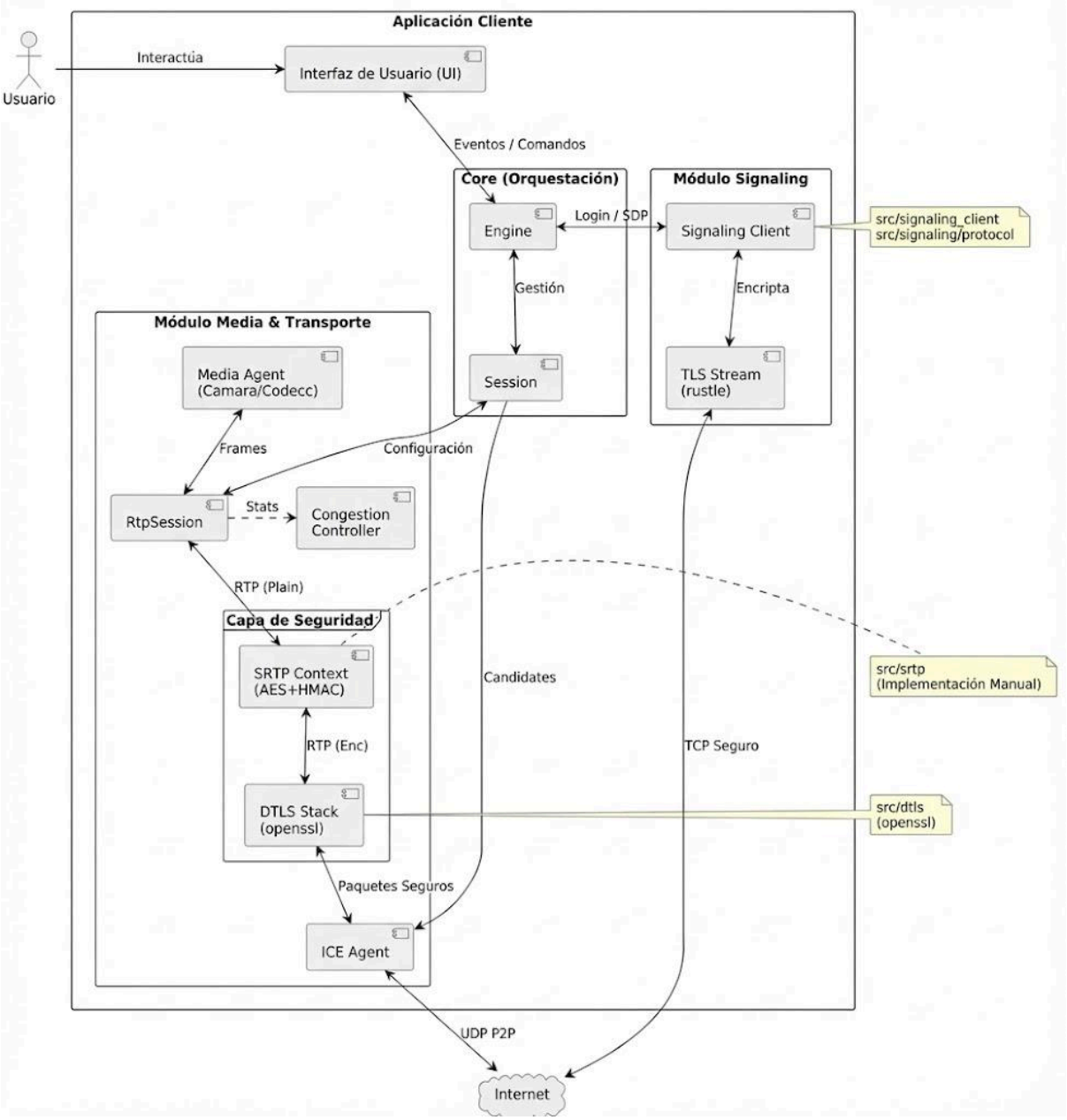

Figura 1.2 - Arquitectura del cliente

A continuación en el informe, iremos explicando detalladamente cada uno de los módulos de dicho diagrama de arquitectura.

Módulo de Señalización (Signaling Server)

¿Qué es y para qué sirve?

El Servidor de Señalización es el punto de encuentro de la red Room RTC. Dado que los clientes no conocen sus direcciones IP entre sí inicialmente, necesitan un intermediario público y confiable para encontrarse. Este módulo, implementado en `src/signaling`, actúa como un servidor TCP concurrente que maneja conexiones persistentes con múltiples clientes. Su responsabilidad principal es autenticar usuarios y enrutar mensajes SDP (Session Description Protocol) entre dos pares que desean comunicarse.

Organización Modular del Servidor

Para garantizar la mantenibilidad y la separación de responsabilidades, el código del servidor se estructuró en módulos funcionales bien definidos. El siguiente diagrama ilustra cómo interactúan las capas de red, la lógica de negocio y la persistencia:

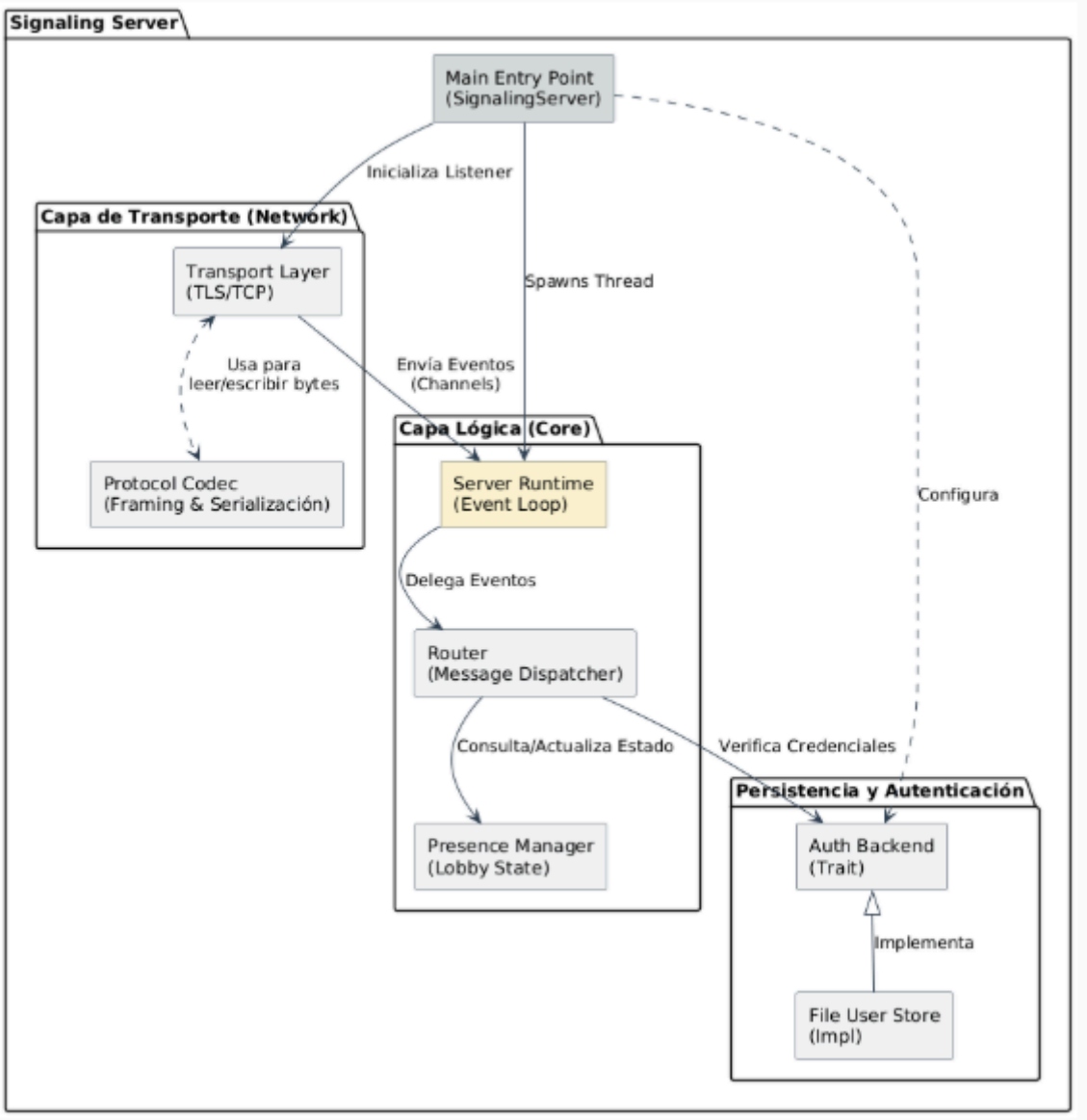

Figura 2.1 - Diagrama de Módulos del Signaling Server

- **Capa de Transporte:** Encapsula la complejidad de la red. Utiliza `rustls` para establecer canales seguros y el `Protocol Codec` para transformar el flujo de bytes TCP en mensajes estructurados del dominio (`SignalingMsg`).
- **Server Runtime & Router:** Constituyen el núcleo (`Core`) de la aplicación. El `Runtime` orquesta el bucle de eventos principal, mientras que el `Router` actúa como el despachador lógico, decidiendo si un mensaje debe ser enrutado a otro cliente, validado contra la base de datos o procesado como un cambio de estado de presencia.
- **Abstracción de Persistencia:** El módulo de autenticación se comunica a través del trait `AuthBackend`. En esta implementación final, se utiliza `FileUserStore` para persistir los datos en disco, pero la arquitectura permite sustituir este componente por una base de datos SQL sin afectar al resto del sistema.

Arquitectura del Servidor y Router

Para manejar múltiples conexiones simultáneas sin bloquear la lógica central, se implementó una arquitectura basada en paso de mensajes (Message Passing) y actores:

1. **Hilo Principal (Acceptor):** Escucha en el puerto TCP configurado. Por cada nueva conexión, envuelve el socket en una capa TLS (usando el crate `rustls`) y lanza un nuevo hilo.
2. **Hilos de Cliente (Connection Threads):** Cada cliente conectado tiene su propio hilo dedicado que:
    - Lee bytes del socket TLS, decodifica el *framing* y envía los mensajes al hilo central mediante un canal MPSC (Multi-Producer, Single-Consumer).
    - Recibe mensajes salientes del hilo central y los escribe en el socket.
3. **Hilo Central (Event Loop & Router):** Es el "cerebro" del servidor. Procesa secuencialmente los eventos de todos los clientes para evitar condiciones de carrera (Race Conditions) en el estado global

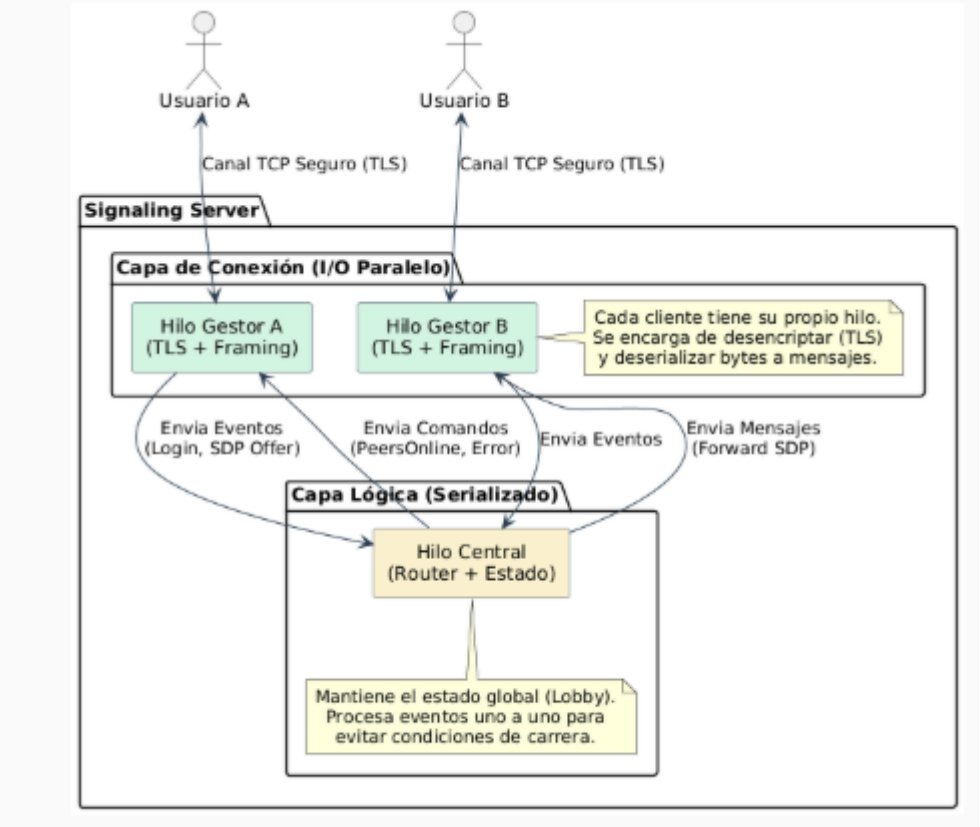

Figura 2.2 - Diagrama de Hilos del Servidor.

Protocolo de Comunicación y Framing

Para la comunicación Cliente-Servidor, diseñamos un protocolo de capa de aplicación personalizado sobre TCP (definido en `src/signaling/protocol`).

Framing (Encuadre): Según la implementación en `framing.rs`, para distinguir mensajes en el flujo de bytes TCP implementamos un esquema de encuadre con cabecera fija. Cada mensaje va precedido por una **cabecera de 8 bytes** con la siguiente estructura:

- **Byte 0:** Versión del protocolo.
- **Byte 1:** Tipo de mensaje (`MsgType`).
- **Bytes 2-3:** Reservados (padding).
- **Bytes 4-7:** Longitud del cuerpo (Payload) en Big Endian.

Esto garantiza que el servidor pueda validar la versión y saber exactamente cuántos bytes leer para procesar el mensaje completo.

Para la carga de certificados y llaves privadas en formato PEM, se utiliza el crate **`rustls-pemfile`**, cuya ruta se define en la configuración. Respecto a la mensajería, aunque el protocolo define tipos como `JoinSession` y `LeaveSession`, actualmente no se utilizan activamente dado que la lógica de negocio se limita a una conexión 1:1, pero quedan disponibles para futuras expansiones a salas grupales.

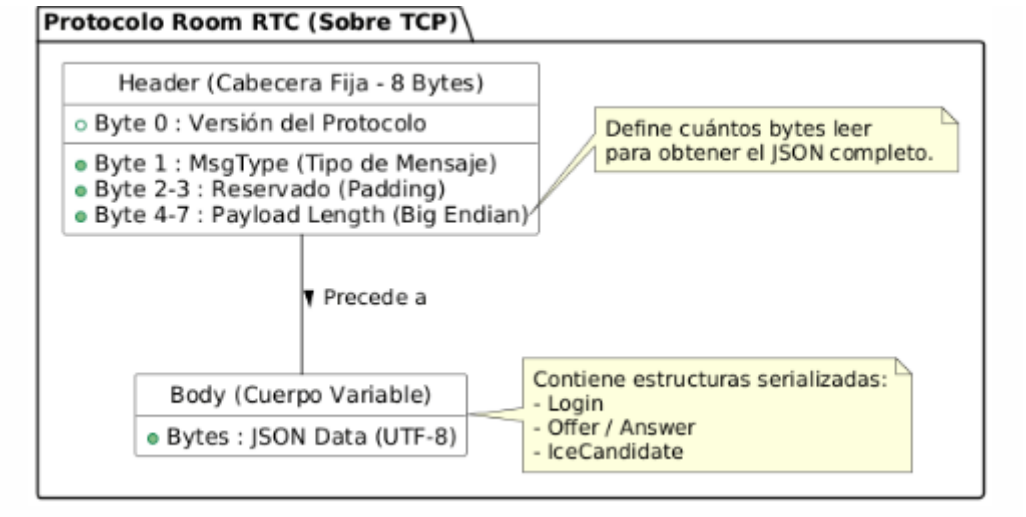

Figura 2.3 - Estructura de Paquetes del Protocolo (Framing).

Tipos de Mensajes

Se definieron estructuras (`Structs`) serializables para cada acción: `Register`, `Login`, `Offer`, `Answer` e `IceCandidate`. Adicionalmente, se implementó el mensaje **`PeersOnline`**, el cual permite al servidor hacer un *broadcast* de la lista actualizada de usuarios cada vez que alguien se conecta, desconecta o entra en llamada con otro usuario.

Gestión de Usuarios y Persistencia (Auth)

Siguiendo los requerimientos, el servidor mantiene una base de datos de usuarios. Para dotar al sistema de flexibilidad, se definió el Trait `AuthBackend`, lo que permite abstraer la implementación de la base de datos (pudiendo ser en memoria, SQL, etc.).

- **Persistencia Atómica:** La implementación concreta `FileUserStore` guarda los usuarios en un archivo de texto plano (`usuario:salt:hash`). Para evitar corrupción de datos, se escribe primero en un archivo temporal (`.tmp`) y luego se realiza un renombrado atómico.
- **Seguridad de Credenciales:** Las contraseñas no se guardan en texto plano. Al registrarse, se genera un Salt aleatorio de 16 bytes. Utilizando el crate `sha2`, se genera un hash SHA-256 combinando el salt y la contraseña, almacenando únicamente el salt y el hash resultante.
- **Configuración:** La ruta del archivo de usuarios es configurable desde el archivo de inicio.

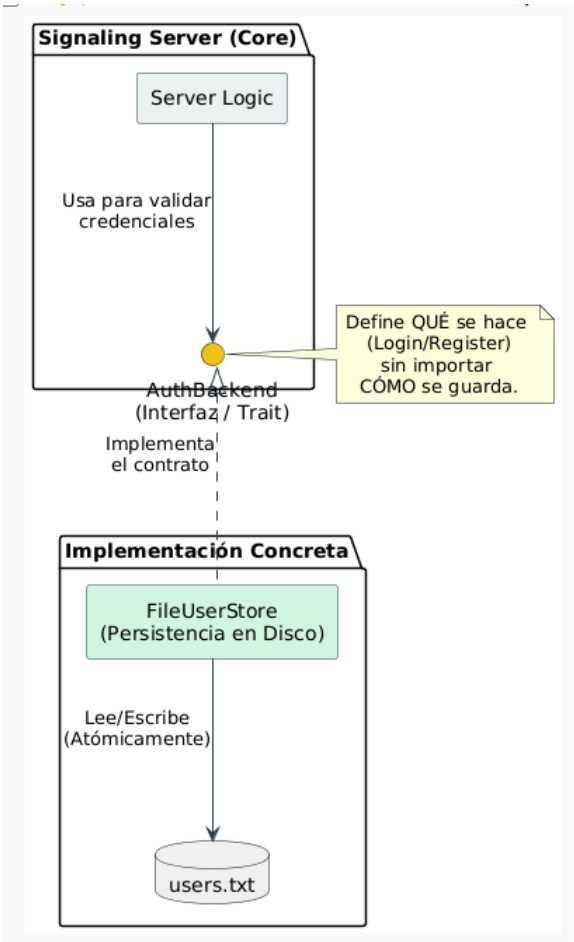

Figura 2.4 - Diagrama Conceptual del Módulo Auth

Manejo de Estados (Presence)

El módulo `presence.rs` gestiona la máquina de estados de cada usuario conectado, permitiendo la lógica del "Lobby":

- **Desconectado:** El usuario no tiene sesión activa TCP.
- **Disponible:** Usuario logueado y listo para recibir llamadas.
- **Ocupado:** Usuario participando activamente en una llamada (el servidor rechaza nuevas ofertas dirigidas a él).

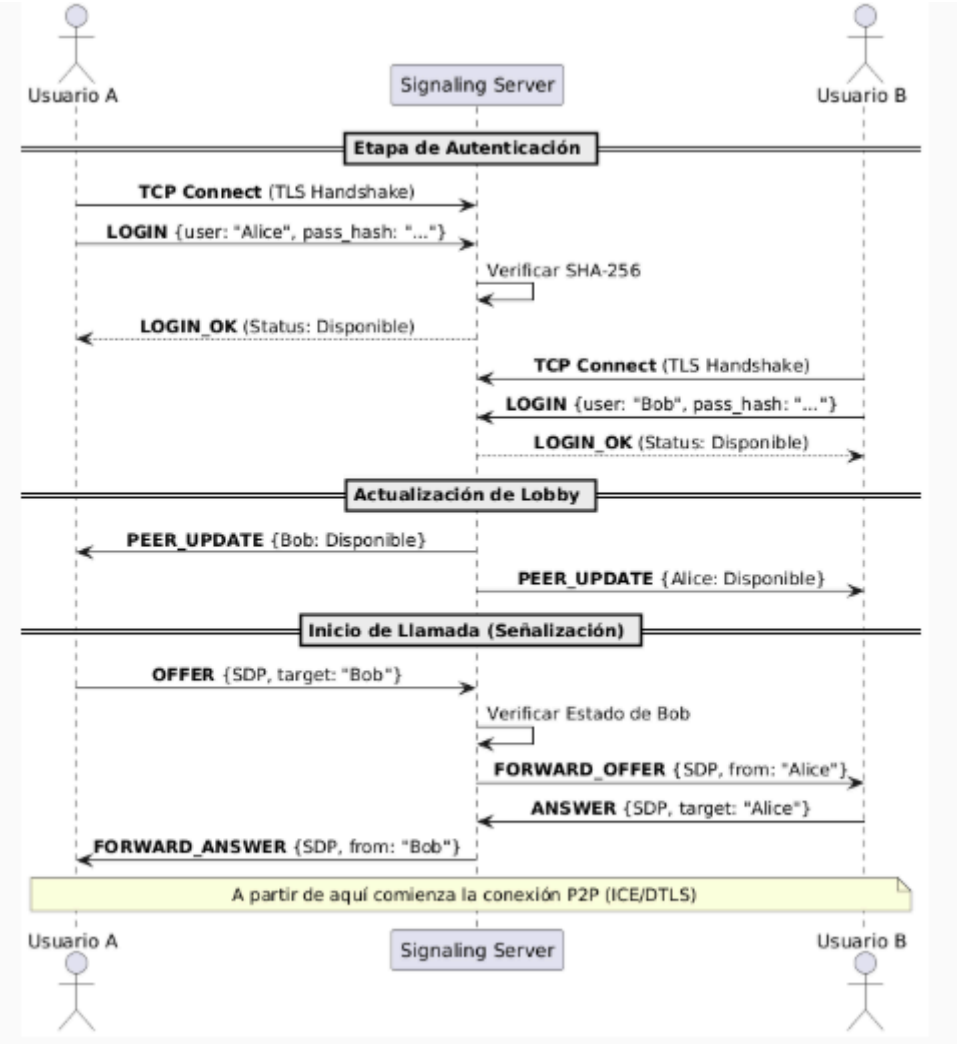

Figura 2.5 - Secuencia de Registro y Llamada (Signaling)

Módulo de Seguridad (Security Layer)

Para la entrega final, se ha implementado una arquitectura de seguridad en capas que protege tanto la señalización (credenciales y metadatos) como el plano de medios (audio/video), cumpliendo con los estándares de WebRTC.

Seguridad en Señalización (TLS)

Toda la comunicación TCP entre el cliente y el Servidor de Señalización está encapsulada en un túnel **TLS (Transport Layer Security)**. Para su implementación se utilizó el crate `rustls`, garantizando que el intercambio de mensajes de registro (`LOGIN`), así como las ofertas y respuestas SDP, viajen cifrados. Esto protege la identidad de los usuarios y evita ataques de *Man-in-the-Middle* durante la fase de negociación.

Intercambio de Claves (DTLS)

Para la conexión P2P UDP, implementamos **DTLS (Datagram TLS)** utilizando bindings de **OpenSSL** (crate `openssl`). Se optó por esta librería debido a su robustez ("battle-tested") y la flexibilidad de configuración que ofrece frente a otras alternativas como `udp_dtls`.

DTLS cumple dos funciones vitales:

1. Realizar un **Handshake** seguro entre los pares. El rol (Client/Server) durante este handshake es determinado dinámicamente por el **IceRole** de la conexión.
2. Derivar las **Llaves Maestras** (Master Keys) utilizando el mecanismo de *Key Exporter* (RFC 5764).

Seguridad en Medios (SRTP)

El componente de mayor complejidad técnica desarrollado en esta etapa es la implementación manual del protocolo **SRTP (Secure Real-time Transport Protocol)**, encapsulado en la estructura `SrtpContext` (`src/srtp/srtp_context.rs`).

A diferencia de soluciones que delegan la criptografía en librerías externas de alto nivel, nuestra implementación maneja explícitamente la transformación de cada paquete RTP. Implementamos el perfil criptográfico **SRTP_AES128_CM_HMAC_SHA1_80**, que define:

- **Cifrado (Confidencialidad):** Utilizamos el algoritmo **AES-128 en modo Contador (CTR)**.
    - Para cada paquete, se calcula un *Initialization Vector (IV)* combinando el *Salt* de la sesión, el SSRC del flujo y el índice del paquete.
    - Se utiliza el crate `aes` y `ctr` para generar un *keystream* que se aplica (XOR) únicamente al *Payload* del paquete RTP, dejando la cabecera legible para que los routers intermedios puedan procesarla.
- **Autenticación e Integridad:** Utilizamos **HMAC-SHA1**.
    - Se firma la totalidad del paquete (Cabecera + Payload Cifrado) junto con el *Rollover Counter (ROC)*.
    - El resultado del hash se trunca a **10 bytes (80 bits)**, siguiendo el estándar, y se anexa al final del paquete como *Authentication Tag*.
    - Al recibir un paquete (`unprotect`), se recalcula este Tag; si no coincide bit a bit con el recibido, el paquete se descarta inmediatamente.
    - Además de `sha1` y `hmac`, se hace uso intensivo del crate **`byteorder`** para la manipulación correcta de la endianness (Big Endian) requerida por los encabezados de red.
- **Protección contra Replay Attacks:** Implementamos una **Ventana de Replay** (`ReplayWindow` en `replay_window.rs`) y lógica de estimación de ROC (`estimate_roc`). El sistema rastrea los índices de los paquetes recibidos recientemente utilizando una ventana deslizante y descarta paquetes duplicados o aquellos que son demasiado antiguos, previniendo que un atacante reenvíe tráfico capturado.

El siguiente diagrama ilustra cómo la estructura `SrtpContext` orquesta las primitivas criptográficas y el estado de la sesión.

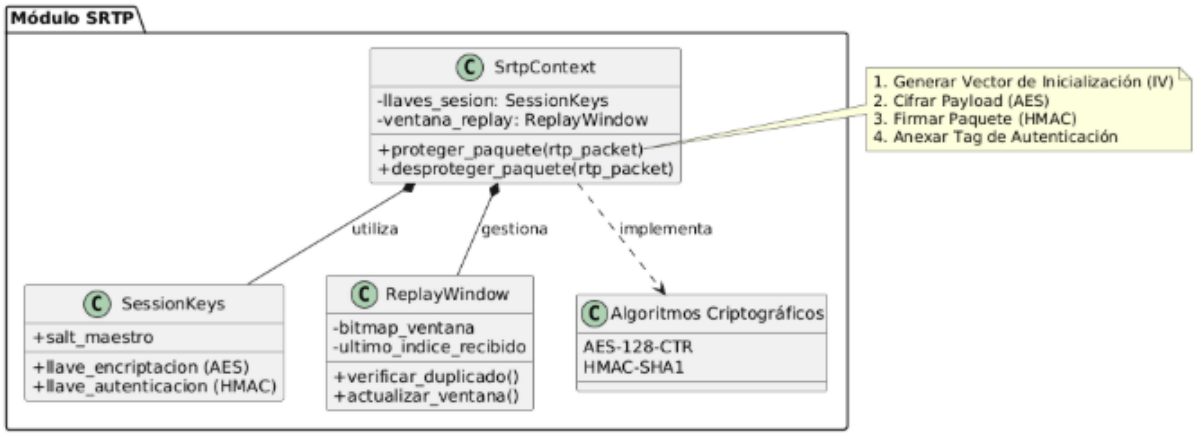

Figura 3.1 - Diseño de Clases del Módulo SRTP

Negociación de Conexión (SDP & ICE)

Módulo ICE

Diseño e Implementación del Agente

El módulo ICE (src/ice) constituye el núcleo de la conectividad P2P. Su responsabilidad es descubrir, validar y mantener la mejor ruta de red posible entre dos pares.

La arquitectura es implementada mediante la interacción entre el IceAgent (lógica pura del protocolo) y el ConnectionManager (integración con SDP y Signaling).

Estrategia de Recolección (Gathering Strategy)

- **Proceso Sincrónico:** Antes de generar una Oferta o Respuesta SDP, el sistema invoca al servicio de *Gathering*.
- **Discovery de Interfaces:** Se inspeccionan las interfaces de red del sistema operativo para identificar direcciones IPv4 locales.
- **Serialización:** Todos los candidatos descubiertos (Host Candidates) se inyectan en el bloque SDP inicial. De esta forma, el mensaje de señalización transporta la "foto completa" de la topología de red del cliente, permitiendo al par remoto iniciar los chequeos inmediatamente tras recibir el SDP.

Algoritmos de Priorización y Emparejamiento

El corazón del agente ICE reside en su capacidad para ordenar deterministicamente los pares de conexión.

- Cálculo de Prioridad: Cada candidato y cada par de candidatos tiene una prioridad asociada. Implementamos la fórmula estándar del RFC para asegurar que ambos extremos prefieran la misma ruta (generalmente la más directa/eficiente):

$$Priority = 2^{32} \times \min(G, D) + 2 \times \max(G, D) + (G > D ? 1 : 0)$$

(Donde G es la prioridad del agente Controlling y D la del Controlled).
- **Formación de Pares (Checklist):** Al recibir los candidatos del par remoto, el agente realiza el producto cartesiano con sus candidatos locales. Estos pares (CandidatePair) se ordenan por prioridad descendente en una *Checklist*, asegurando que las rutas más prometedoras (ej. LAN vs LAN) se verifiquen primero.

Verificación de Conectividad (Connectivity Checks)

La validación de rutas se ejecuta de forma asíncrona para no bloquear el hilo principal de la aplicación.

- **IceWorker:** Se implementó un *worker* en segundo plano (spawn_ice_worker en connection_manager.rs) encargado del I/O de red.
- **Protocolo STUN:** El worker itera sobre la *Checklist* enviando mensajes **STUN Binding Request** a través de UDP.
- **Máquina de Estados del Par:** Cada par evoluciona individualmente:
  - Waiting: Estado inicial.
  - InProgress: Se envió un Binding Request, esperando respuesta.
  - Succeeded: Se recibió un Binding Response válido. El par es viable.
  - Failed: Timeout o error de red.

Gestión de Roles (Controlling vs. Controlled)

Para resolver conflictos (por ejemplo, si ambos pares intentan nominar rutas distintas), el sistema asigna roles lógicos basados en la señalización:

- **Controlling:** Generalmente el iniciador de la llamada (quien envía la Oferta). Tiene la potestad final de **Nominar** el par ganador marcándolo con el flag USE-CANDIDATE.
- **Controlled:** El receptor (quien envía la Respuesta). Actúa pasivamente esperando la nominación.

Transición al Plano de Datos Seguro

Una vez que el agente Controlling nomina un par válido (Succeeded), el proceso de ICE concluye.

El socket UDP asociado a ese par "ganador" no se cierra; por el contrario, se promueve y se transfiere al módulo DTLS. A partir de este momento, cualquier paquete enviado por ese socket será parte del handshake de seguridad o tráfico SRTP (video), completando el establecimiento de la sesión.

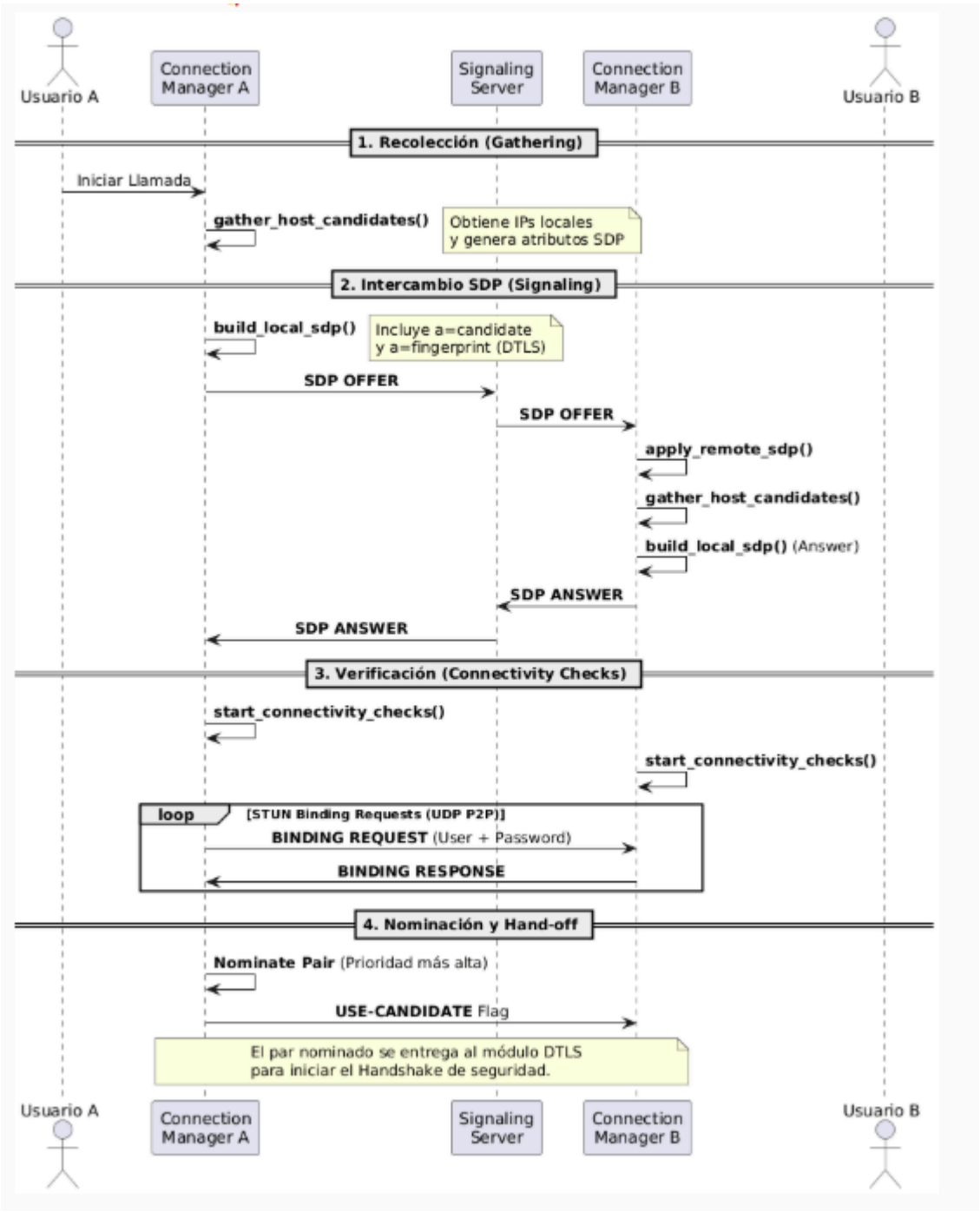

Figura 4.1 - Secuencia de Negociación ICE

STUN(Session Traversal Utilities for NAT)

Introducción

En una comunicación P2P real, como la que utiliza WebRTC, uno de los principales desafíos es que la mayoría de los usuarios se encuentran detrás de un NAT (Network Address Translator). Esto implica que la dirección IP que cada cliente usa dentro de su red local (por ejemplo, 192.168.x.x o 10.x.x.x) no es visible desde Internet, por lo que un peer externo no puede contactarlo directamente utilizando su dirección privada. Para resolver este problema, el protocolo WebRTC utiliza STUN, definido en el RFC 5389.

¿Qué es?

STUN es un protocolo ligero cuya función es permitir que un cliente pueda descubrir su dirección pública —la que realmente ve Internet— así como el puerto externo que su NAT le asignó. Esta dirección pública se denomina server reflexive address.

Por ejemplo: Si un cliente está dentro de una LAN con IP 192.168.0.10, STUN le responde, “Tu IP pública es 181.31.55.120 y tu puerto externo es 58234".

Esta información es esencial para que otros peers puedan contactarlo directamente, permitiendo establecer conexiones P2P incluso si ambos están detrás de NAT.

¿Cómo se integra en el flujo de ICE?

Dentro del proceso de ICE (Interactive Connectivity Establishment), STUN es responsable de generar uno de los tipos más importantes de candidatos: los server reflexive (srflx). De esta forma se generan host y server reflexive candidates, mientras que los host candidates solo sirven dentro de la LAN, los reflexive permiten intentar conexiones a través de Internet.

Uso de un servidor STUN público

Para obtener esta dirección pública, nuestro proyecto consulta un servidor STUN público, en particular:

stun.l.google.com:19302

El cual fue configurado por properties, en el proyecto.

Estos servidores responden a Binding Requests enviados por el cliente y devuelven la dirección pública desde la cual vieron el paquete UDP. Esto nos permite obtener el server reflexive candidate, que luego se incorpora al set de candidatos ICE.

Pasos de la implementación:

- A la funcionalidad “gather_candidates”, que se encuentra en ice_agent, además de consultar los candidatos locales, se incorporó una funcionalidad “gather_stun_candidates”, que consulta los candidatos al server de google.
- Dentro de esta función, la app se conecta al servidor de google, y se obtiene un candidato de tipo “server reflexive”.
- De esa manera ice_agent, ya cuenta con 2 tipos de candidatos, entonces en el próximo paso se hace un producto cartesiano de todos, y se continúa con los siguientes pasos del proceso del protocolo ICE.

Módulo SDP

Visión General y Rol en la Arquitectura

El módulo SDP (`src/sdp`) actúa como la capa de traducción entre el estado interno de la aplicación. En la arquitectura final de "Room RTC", el SDP cumple un rol crítico de integración, transportando tres tipos de información vital:

1. **Negociación de Medios:** Definición de códecs (OpenH264), Payload Types y frecuencias de reloj.
2. **Información de Conectividad (ICE):** Transporte de los candidatos de red (`a=candidate`) y credenciales (`ufrag`, `pwd`).
3. **Información de Seguridad (DTLS):** Transporte del **Fingerprint SHA-256** del certificado local (`a=fingerprint`), necesario para autenticar el handshake DTLS.

Diseño del Modelo de Datos

Para manejar la complejidad jerárquica del protocolo, se diseñó un modelo de datos que refleja la estructura lógica del estándar, priorizando la tipificación fuerte sobre el manejo de cadenas crudas:

- **Estructura Raíz (`Sdp`):** Representa la sesión completa. Contiene los metadatos globales (versión, origen, nombre de sesión) y una lista de bloques de medios.
- **Bloques de Medios (`Media`):** Modela cada flujo (Stream) de la sesión (ej. "video"). Encapsula el puerto, protocolo (`RTP/SAVPF` para indicar SRTP sobre DTLS) y formatos.
- **Atributos (`Attribute`):** Una estructura flexible clave-valor que modela las líneas `a=`. Se utiliza tanto para atributos estándar (`rtpmap`, `fmtp`) como para extensiones necesarias para ICE y DTLS (`candidate`, `fingerprint`, `setup`).

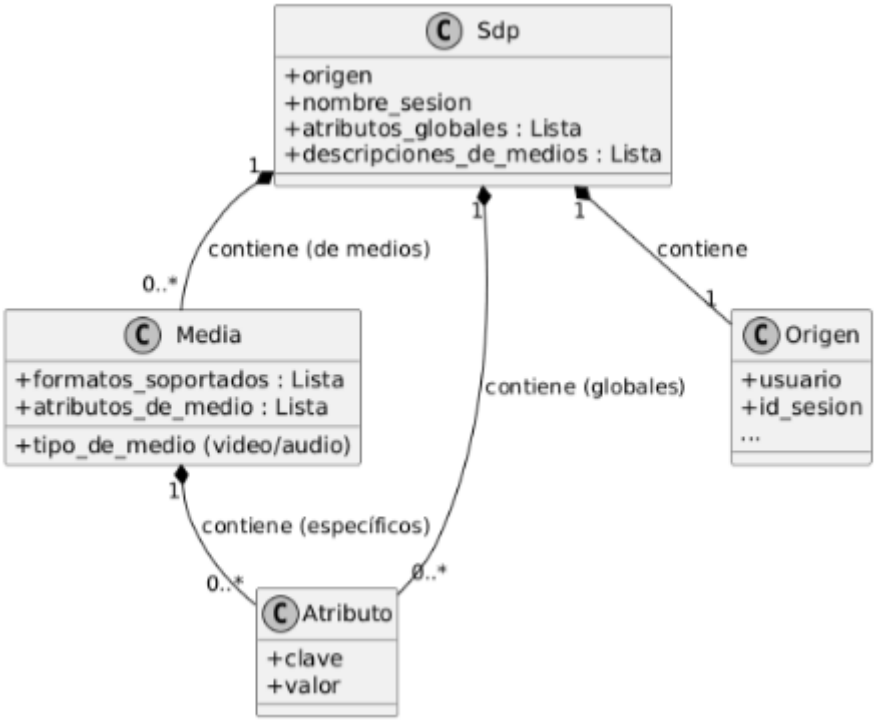

Figura 4.2 - Diagrama de Estructura del Módulo SDP

Motor de Serialización y Parseo

En lugar de utilizar expresiones regulares frágiles, implementamos un motor de parseo robusto:

- **Deserialización (Parseo):** El parser recorre el texto línea por línea, detectando los prefijos de campo (`m=`, `a=`, etc.). Mantiene un contexto interno que determina si un atributo pertenece a la sesión global o al último bloque de medios detectado, poblando las estructuras `Sdp` y `Media` dinámicamente.
- **Serialización (Generación):** El proceso inverso delega la responsabilidad en cada estructura. El objeto `Sdp` orquesta la escritura de las cabeceras y luego itera sobre sus objetos `Media`, los cuales a su vez serializan sus listas de `Attribute`. Esto asegura que el SDP generado siempre esté bien formado y cumpla con el orden estricto requerido por el RFC.

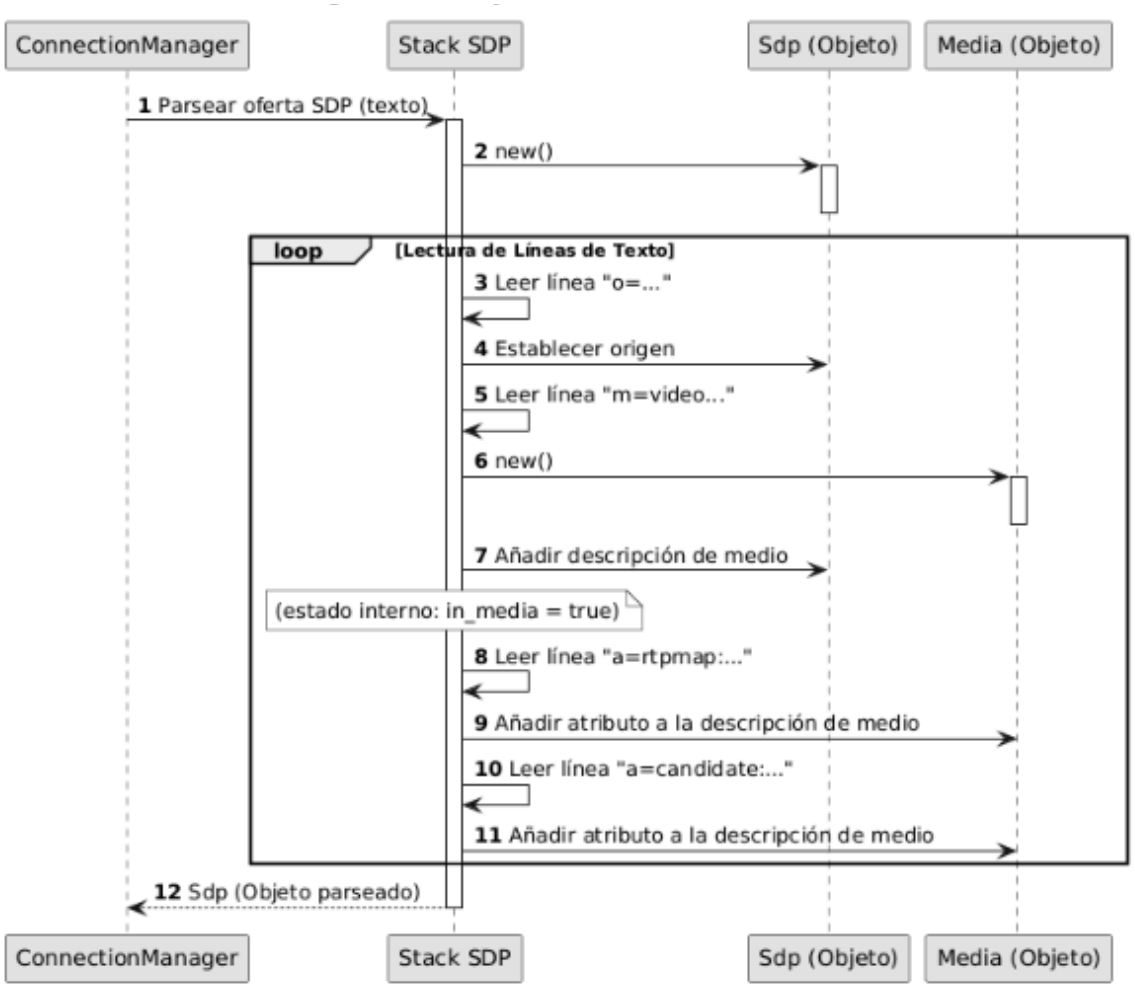

Figura 4.3 - Diagrama de Secuencia del Flujo de Parseo de Oferta

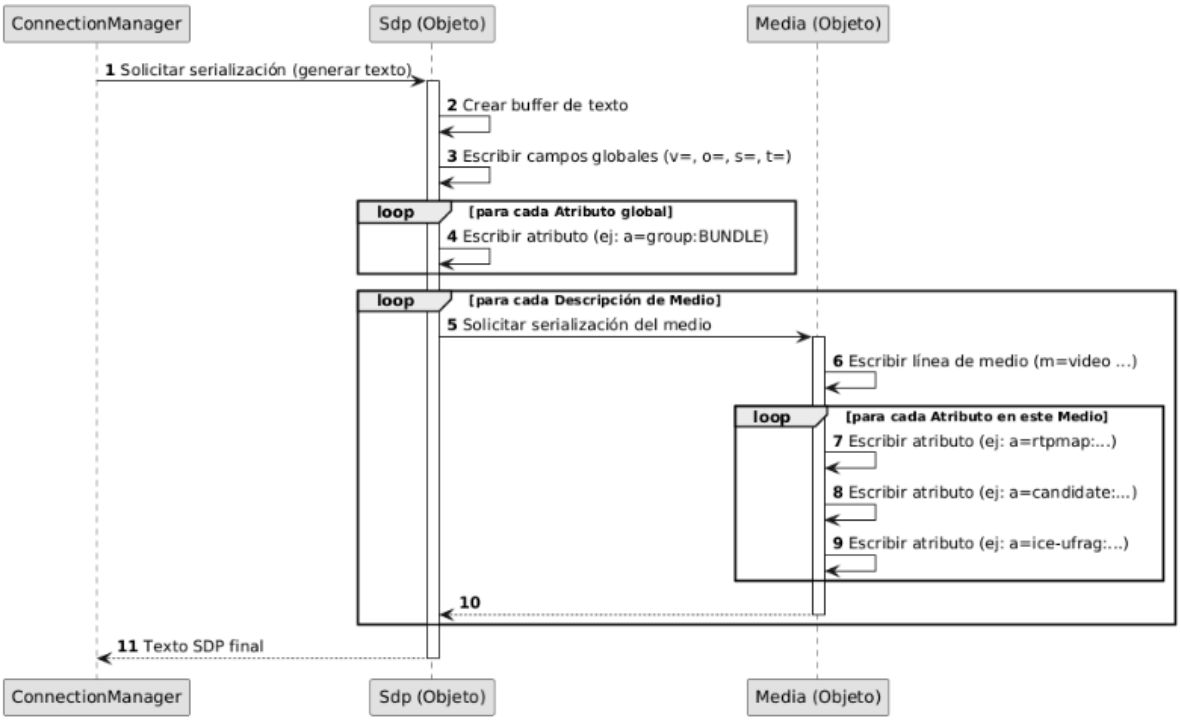

Figura 4.4 - Diagrama de Secuencia (Lógica de Serialización Detallada)

Integración con el ConnectionManager

El `ConnectionManager` utiliza este módulo como una librería de utilidades pura. En el flujo de una llamada (ver Figura 4.1), el Manager:

1. Recolecta los **Candidatos ICE** y el **Fingerprint DTLS**.
2. Construye programáticamente un objeto `Sdp`.
3. Invoca al serializador para obtener el texto (`String`) que se enviará al **Signaling Server**.
4. Al recibir una respuesta, utiliza el parser para obtener un objeto `Sdp` y extrae de él las IPs remotas y el Fingerprint del par para configurar el transporte seguro.

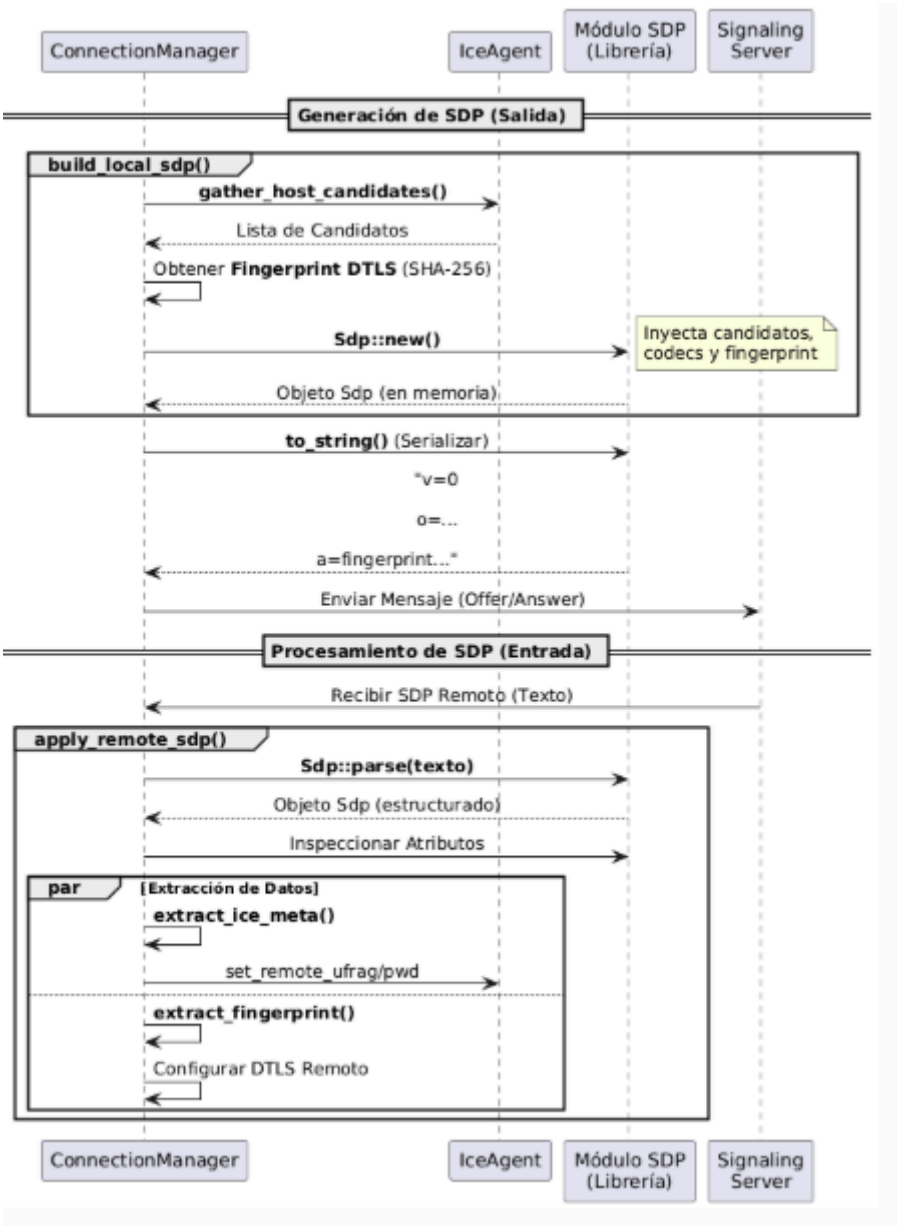

Figura 4.5 - Integración SDP con ConnectionManager

Módulo de Medios y Transporte (Media Stack)

Selección del Códec de Video

OpenH264 Para la transmisión de video, el sistema utiliza el estándar **H.264**. Durante la fase de investigación evaluamos el uso de la librería `x264` (común en implementaciones profesionales), pero finalmente optamos por **OpenH264**, la implementación de código abierto de Cisco.

Esta decisión técnica se fundamenta en los siguientes criterios:

- **Compatibilidad WebRTC:** Cumple estrictamente con el RFC 6184 y facilita la integración con SDP e ICE.
- **Estabilidad en Rust:** El crate `openh264` (v0.3) ofrece bindings modernos y seguros para Rust, a diferencia de `x264` que presentó problemas de mantenimiento.
- **Portabilidad:** Al ser una implementación ligera en C++, su compilación y despliegue multiplataforma resultaron más sencillos.
- **Interoperabilidad:** Es el mismo motor utilizado internamente por Firefox y Cisco Webex, lo que garantiza compatibilidad con clientes externos.

Desventaja asumida: Somos conscientes de que OpenH264 puede tener una eficiencia de CPU levemente inferior a `x264` en entornos de alta exigencia, pero la estabilidad ganada compensa este factor para el alcance del proyecto.

Capa de Transporte de Medios (Media Transport)

Este módulo (`src/media_transport`) actúa como el puente crítico entre la lógica de aplicación (Frames) y la red (Paquetes). Su función es adaptar los datos audiovisuales para su transmisión eficiente y ordenada.

El diseño se estructura en dos pipelines paralelos que operan de forma independiente para maximizar el rendimiento:

1. **Pipeline de Salida (Packetizer):** Fragmenta los frames comprimidos (NAL Units) en paquetes RTP.
    - **Lógica de Fragmentación:** Implementamos un `H264Packetizer` que inspecciona el tamaño de cada NAL Unit. Si una unidad excede el MTU de la red (Maximum Transmission Unit), se aplica el mecanismo de fragmentación **FU-A (Fragmentation Unit Type A)** definido en el RFC 6184. Esto divide el frame en múltiples paquetes RTP más pequeños, añadiendo las cabeceras de secuencia necesarias (Start/End bits) para su posterior reensamblado.
2. **Pipeline de Entrada (Depacketizer):** Reconstruye el flujo de video a partir de los paquetes RTP entrantes.
    - **Lógica de Reensamblado:** El `H264Depacketizer` mantiene un buffer de estado temporal. Al recibir paquetes fragmentados (FU-A), los acumula ordenadamente hasta detectar el bit de "End". Una vez que se tiene la unidad completa o se recibe un paquete con el marcador `Marker Bit=1`, se emite el frame completo hacia el decodificador.

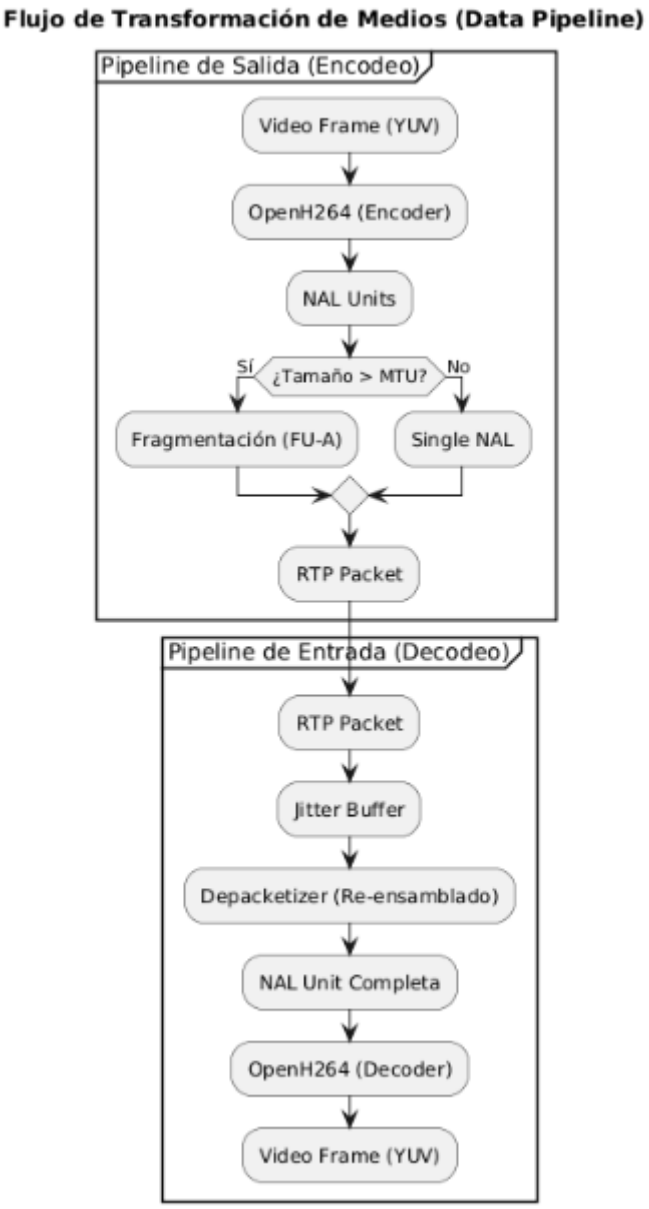

Figura 5.1 - Flujo de Transformación de Video (Packetization Flow)

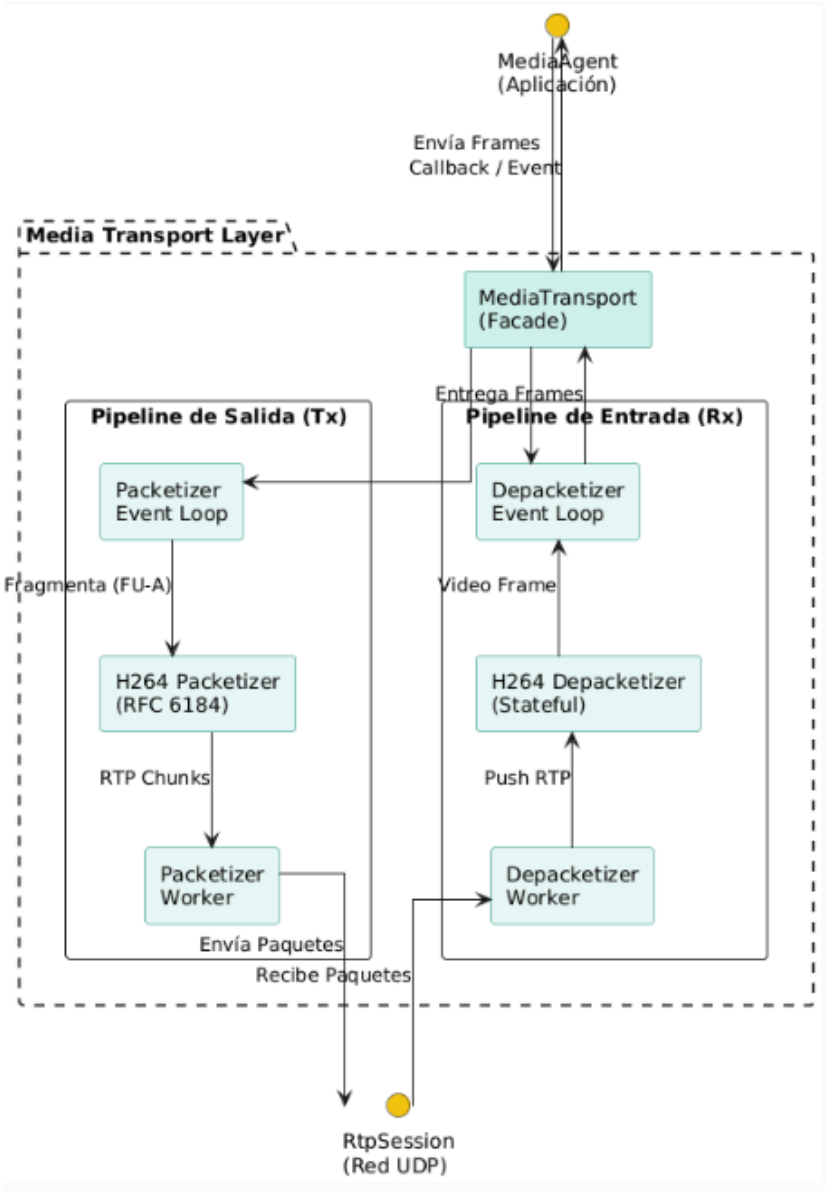

Figura 5.2 - Diagrama de Submódulos del Media Transport

Arquitectura del MediaAgent

El orquestador del procesamiento audiovisual (`src/media_agent`) coordina la captura, codificación y visualización. A diferencia de versiones preliminares que utilizaban un ciclo de reloj fijo ("tick") para forzar el envío de frames, la arquitectura final implementa un **modelo reactivo basado en eventos y bitrate**.

- **Codificación por Demanda:** El hilo de captura de cámara (`CameraWorker`) no genera frames indiscriminadamente, sino que recibe un parámetro `target_fps` al momento de su creación. La velocidad de captura se ajusta a esta configuración, asegurando un control preciso sobre la tasa de generación de frames antes de enviarlos al `EncoderWorker`.

- **Control de Flujo:** El codificador no trabaja a ciegas; respeta estrictamente el *bitrate* configurado por el controlador de congestión. Si la red se satura, el codificador reduce la calidad o descarta frames proactivamente antes de generar los paquetes, evitando el bufferbloat en la red.
- **Eficiencia de Recursos (Sleeps & Timeouts):** Tanto el `EncoderWorker` como el `DecoderWorker` implementan una lógica estricta de timeouts y sleeps en sus bucles de procesamiento. Si no hay frames pendientes o eventos entrantes, los hilos entran en estado de espera (sleep), evitando bucles activos (busy-loops) y minimizando drásticamente el uso de CPU cuando el sistema está en reposo.

Flujo de Eventos y Mensajería

La comunicación interna se basa en el paso de mensajes asíncronos coordinados por el `MediaAgent Listener`:

- **Flujo Remoto (Rx):** Los eventos `AnnexBFrameReady` (video comprimido) son entregados por el `MediaTransport` al hilo principal del agente. Este los deriva al `DecoderWorker`. Una vez procesado, el worker emite un evento `DecodedVideoFrame` de vuelta al listener a través de un canal, actualizando el `remote_frame` para que el Engine pueda consumirlo mediante `snapshot_frames()`.
- **Flujo Local (Tx):** De forma análoga, los frames capturados por el `CameraWorker` se envían al listener, quien los delega al `EncoderWorker`. Al finalizar la compresión, se levanta el evento `EncodedVideoFrame`, que el listener redirige finalmente al `MediaTransport` para su packetización.

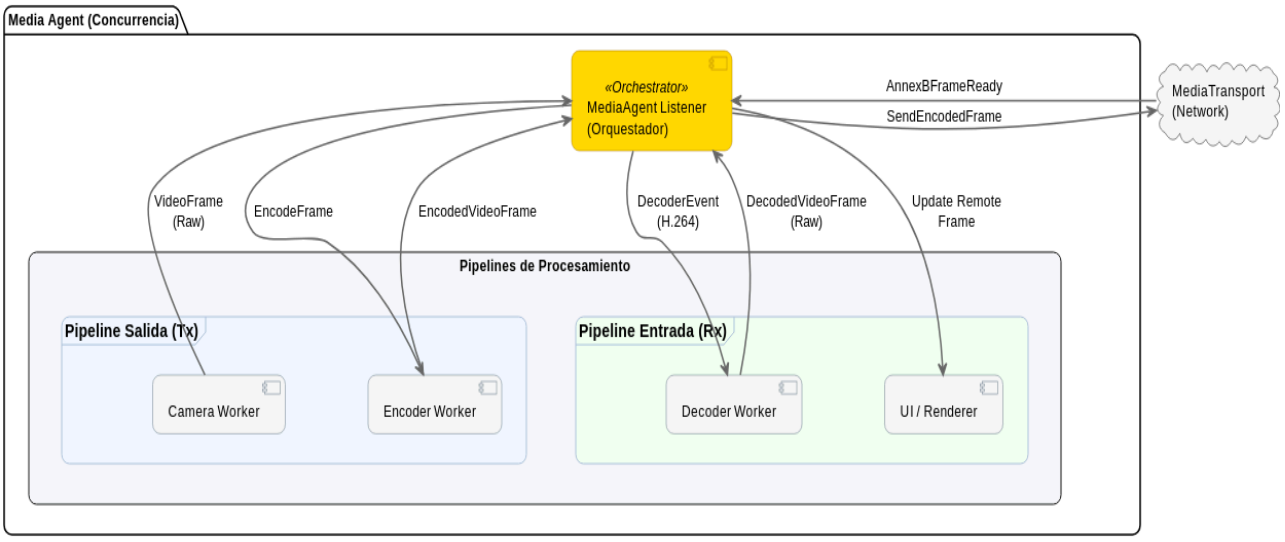

Figura 5.3 - Diagrama de Hilos del Media Agent

Protocolos de Transporte (RTP y RTCP)

La capa de transporte (`src/rtp_session`) implementa la lógica de red en tiempo real.

Modelado de Datos (Structs Principales)

Para garantizar un control total sobre los bits que viajan por la red, modelamos las estructuras de la siguiente manera:

- **`RtpPacket`:** Encapsula la cabecera (versión, padding, extensión) y el payload.
- **`RtpRecvStream` / `RtpSendStream`:** Estructuras que mantienen el estado de cada flujo (secuencia, jitter, timestamps). Permiten manejar pérdidas y reordenamiento de paquetes.

Control y Feedback (RTCP)

Implementamos un ciclo de retroalimentación robusto para mantener la calidad de la llamada:

- **Sender/Receiver Reports (SR/RR):** Para sincronización de labios (Lip-sync) y cálculo de RTT.
- **NACK (Negative Acknowledgement):** Gestión de retransmisiones ante pérdida de paquetes.
- **PLI (Picture Loss Indication):** Si el decodificador no puede reconstruir la imagen (por pérdida de un paquete crítico), solicita un *Keyframe* completo al emisor.

Control de Congestión

Incorporamos un módulo de **Control de Congestión** (`src/congestion_controller`) que actúa como un lazo de control cerrado para adaptar la calidad del video al estado de la red. El proceso se detalla a continuación:

- **Cálculo de Métricas (`NetworkMetrics`):** El sistema analiza los paquetes RTCP entrantes (Receiver Reports) para extraer estadísticas clave como la pérdida de paquetes y el RTT. Estos datos se consolidan en la estructura `NetworkMetrics`.
- **Evento de Ajuste (`UpdateBitrates`):** Basándose en estas métricas, el controlador decide si es necesario subir o bajar la calidad. Si detecta congestión, emite un evento `UpdateBitrates`.
- **Actuación Dinámica:** Este evento es consumido tanto por el **Media Agent** como por el **Media Transport**:
    - El **Media Agent** reconfigura el encoder `OpenH264` al vuelo para reducir el bitrate objetivo de los nuevos frames.
    - El **Media Transport** ajusta su ritmo de envío, evitando saturar el canal con ráfagas de paquetes.

Integración de Módulos (Pipeline)

La siguiente figura ilustra cómo fluyen los datos a través de las capas de la aplicación, desde la captura hasta la red.

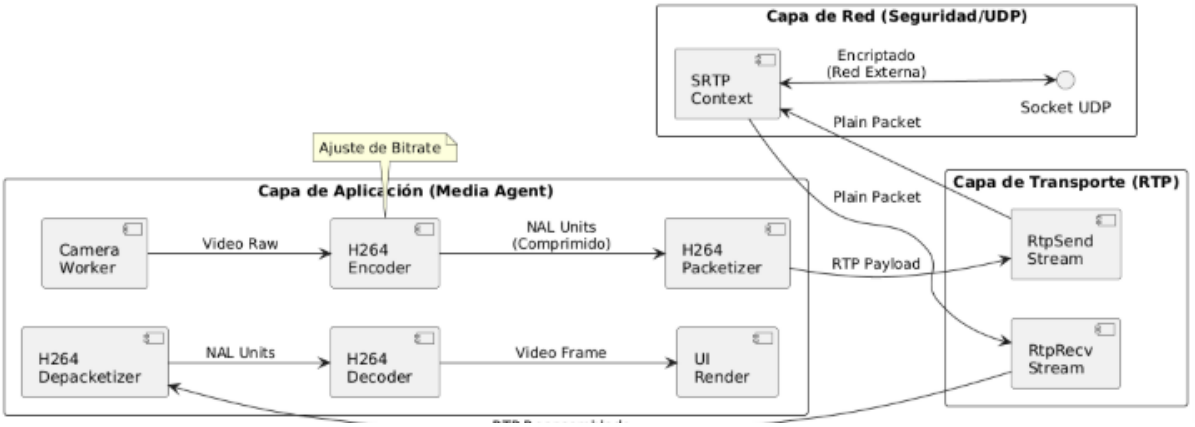

Figura 5.4 - Pipeline de Procesamiento y Transporte de Video

Flujo Integrado de la Aplicación (End-to-End)

Finalmente, presentamos el flujo completo de una sesión, integrando el Servidor de Señalización, la Negociación y el Establecimiento de Medios Seguros.

El siguiente diagrama de secuencia detalla la interacción desde el inicio de sesión hasta la transmisión de video encriptado.

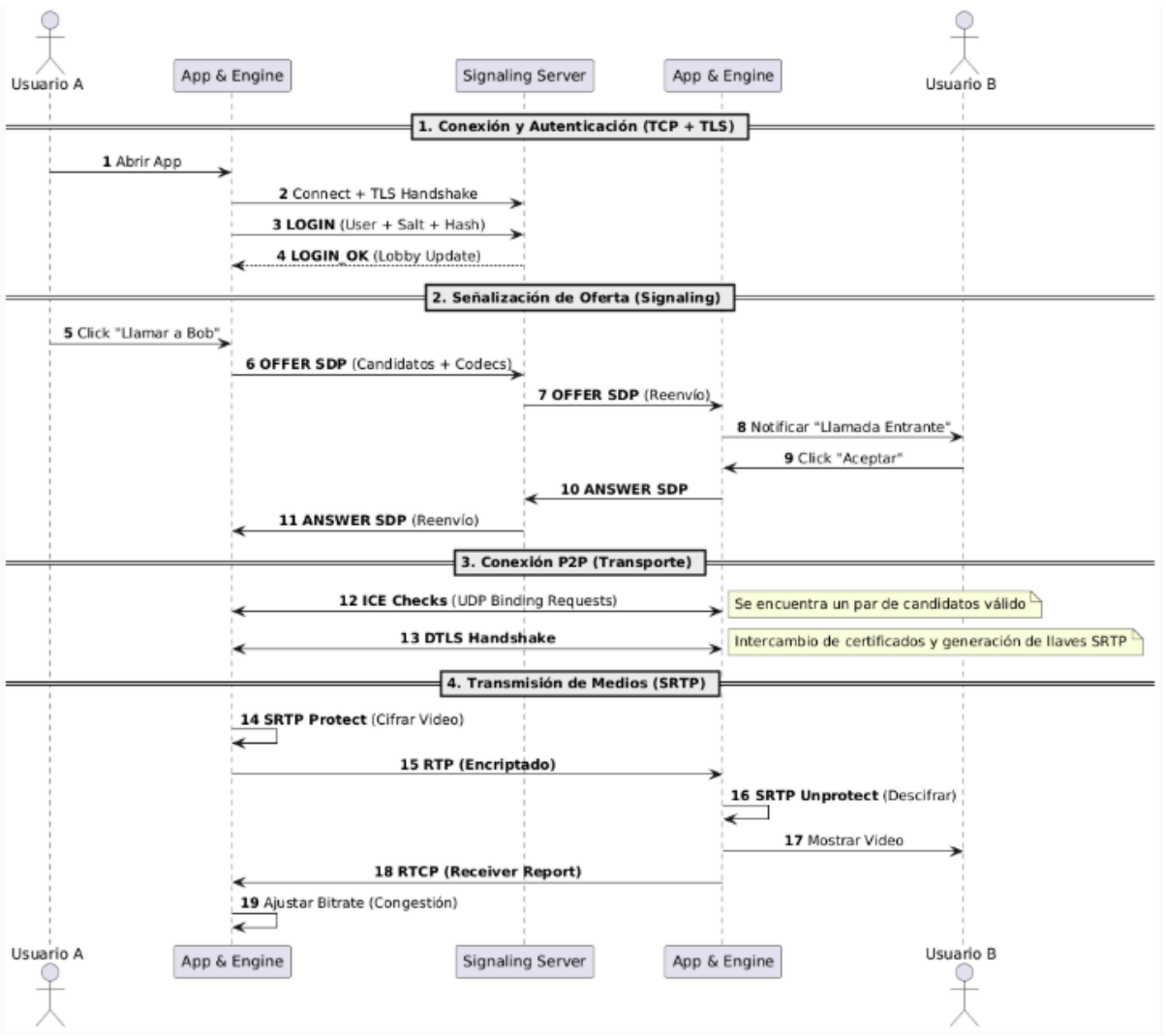

Figura 6.1 - Secuencia Completa de Establecimiento de Llamada

Conclusiones y aprendizajes

La realización del proyecto "Room RTC" ha sido una experiencia sumamente enriquecedora, permitiéndonos no solo cumplir con un objetivo académico, sino enfrentar desafíos de ingeniería reales. A lo largo del cuatrimestre, logramos implementar paso a paso cada componente de un sistema de videoconferencias, comprendiendo en profundidad la complejidad oculta detrás de la comunicación en tiempo real.

Logros Técnicos

Hemos logrado evolucionar el proyecto desde una arquitectura local básica hasta un sistema distribuido y seguro. La implementación manual de protocolos criptográficos de bajo nivel (como la transformación de paquetes RTP a SRTP) y el desarrollo de un protocolo de señalización propio sobre TCP validaron nuestra capacidad para construir soluciones robustas en Rust, priorizando la seguridad y el rendimiento.

Beneficios del Diseño Arquitectónico

La arquitectura en capas (UI <> Engine <> Session <> Transport) demostró facilitar el desarrollo, permitiendo probar módulos críticos como RtpSession en aislamiento (Testabilidad) y simplificando la incorporación futura de nuevas características, como canales de audio, sin reescribir la lógica central (Extensibilidad).

Metodología y Trabajo en Equipo

- **Gestión Ágil:** La división del trabajo en **sprints semanales** nos permitió avanzar de manera incremental y ordenada, facilitando la integración de módulos desarrollados por distintos miembros del equipo.
- **Herramientas:** La adopción de herramientas como **GitHub Projects** fue clave para la distribución de tareas y el seguimiento del progreso.
- **Feedback Docente:** La guía constante y el feedback semanal de nuestro tutor fueron determinantes para orientar el diseño de la arquitectura y corregir desvíos a tiempo.

Investigación y Nuevas Herramientas

El proceso de investigación fue muy fructífero. Nos vimos impulsados a aprender y dominar nuevas herramientas y librerías que no conocíamos, como la integración del códec **OpenH264** o el manejo de criptografía con primitivas básicas. Esta necesidad de investigar cada componente antes de implementarlo nos brindó un aprendizaje técnico profundo que va más allá de la simple utilización de librerías.

Agregado de Funcionalidades

Aquí tienes la redacción adaptada específicamente para encajar en tu **Informe Final**, imitando el tono técnico, la estructura y el vocabulario utilizado en la sección de transferencia de archivos (FileHandler) que se observa en tu documento.

Esta sección está lista para ser copiada e insertada en tu informe (probablemente antes o después de la sección de Transmisión de Archivos).

Transmisión de Audio en Tiempo Real

Diseño e Implementación

Para la incorporación de audio en el stack rusty-rtc, se diseñó un subsistema modular que prioriza la baja latencia y la eficiencia computacional, evitando dependencias complejas de serialización. La arquitectura separa las responsabilidades de interacción con el hardware (I/O) de la lógica de orquestación y transporte.

Selección de Códec: G.711 μ-law

Dada la restricción de no utilizar librerías externas pesadas y la necesidad de una implementación "simple pero eficiente", se optó por implementar el estándar **G.711** (audio_codec.rs) en lugar de códecs más modernos pero complejos como Opus.

- **Eficiencia:** G.711 ofrece una compresión de 2:1 (16-bit PCM a 8-bit) con un costo de CPU despreciable, ideal para el entorno de laboratorio.
- **Formato:** Se trabaja con una frecuencia de muestreo de 8000 Hz (banda estrecha) y canales mono, generando un flujo de datos constante y predecible de 64 kbps.

Modelo de Concurrencia

El manejo de audio es extremadamente sensible a bloqueos. Para garantizar que la interfaz gráfica (UI) y el transporte de red no introduzcan *jitter* o cortes, se implementó un modelo de **hilos dedicados (Workers)** conectados mediante canales mpsc:

1. **AudioCaptureWorker:** Hilo de alta prioridad encargado exclusivamente de interactuar con el driver de audio (cpal) para la lectura del micrófono.
2. **AudioPlayerWorker:** Hilo independiente que gestiona el buffer de salida y la escritura en los parlantes.
3. **MediaAgent:** Actúa como orquestador central, puenteando los datos entre los workers de audio y el MediaTransport (RTP).

Estrategia de Muteo (Soft Mute)

Una decisión de diseño clave fue la implementación del "Soft Mute". En lugar de detener el envío de paquetes RTP cuando el usuario silencia su micrófono, el sistema continúa transmitiendo tramas que contienen silencio digital (ceros).

- **Justificación:** Esto mantiene la continuidad en los números de secuencia y *timestamps* del protocolo RTP. Si se dejara de transmitir, el receptor podría interpretar el silencio como una pérdida masiva de paquetes o una desconexión, desestabilizando el buffer de reproducción.
- **Mecanismo:** Se utiliza un AtomicBool compartido entre el MediaAgent y el CaptureWorker para una conmutación instantánea y segura entre hilos.

Flujo End-to-End

Captura y Envío (Sender Flow)

El proceso inicia en el AudioCaptureWorker, el cual despierta periódicamente impulsado por el reloj de audio del sistema operativo.

1. **Captura y Chunking:** Se leen las muestras crudas (f32) del dispositivo de entrada. Estas se acumulan hasta formar un bloque de **160 muestras** (equivalente a 20ms de audio, un estándar en VoIP).
2. **Verificación de Estado:** Antes de procesar, se consulta el estado de muteo. Si está activo, el buffer se inyecta con ceros; de lo contrario, se procesa el audio capturado.

3. **Codificación:** El bloque se comprime utilizando el algoritmo μ-law, resultando en un *payload* de 160 bytes.
4. **Transporte:** El MediaAgent recibe la trama comprimida y la deriva al MediaTransport. Allí, se encapsula en un paquete RTP (con *Payload Type* específico para PCMU) y se envía a la red vía UDP/DTLS.

Recepción y Reproducción (Receiver Flow)

Los paquetes RTP entrantes son interceptados por el MediaTransport del nodo remoto.

1. **Depacketization:** Se extrae el *payload* de audio y se verifica la secuencia RTP para detectar pérdidas.
2. **Decodificación:** El MediaAgent recibe los bytes comprimidos e invoca al audio_codec para expandirlos nuevamente a muestras PCM (Vec<f32>).
3. **Buffering y Jitter:** Las muestras decodificadas se envían al AudioPlayerWorker. Este hilo mantiene una cola (VecDeque) que actúa como *Jitter Buffer* rudimentario.
4. **Control de Latencia:** Si la red se congestiona y luego entrega una ráfaga de paquetes, el buffer podría crecer demasiado, introduciendo latencia. El AudioPlayerWorker implementa una lógica de descarte: si la cola supera un umbral crítico (MAX_BUFFER_SIZE), se descartan las muestras más antiguas para "alcanzar" el tiempo real.

Diagramas de Diseño

A continuación se presentan los diagramas que detallan la interacción para el flujo de audio y el mecanismo de control de muteo:

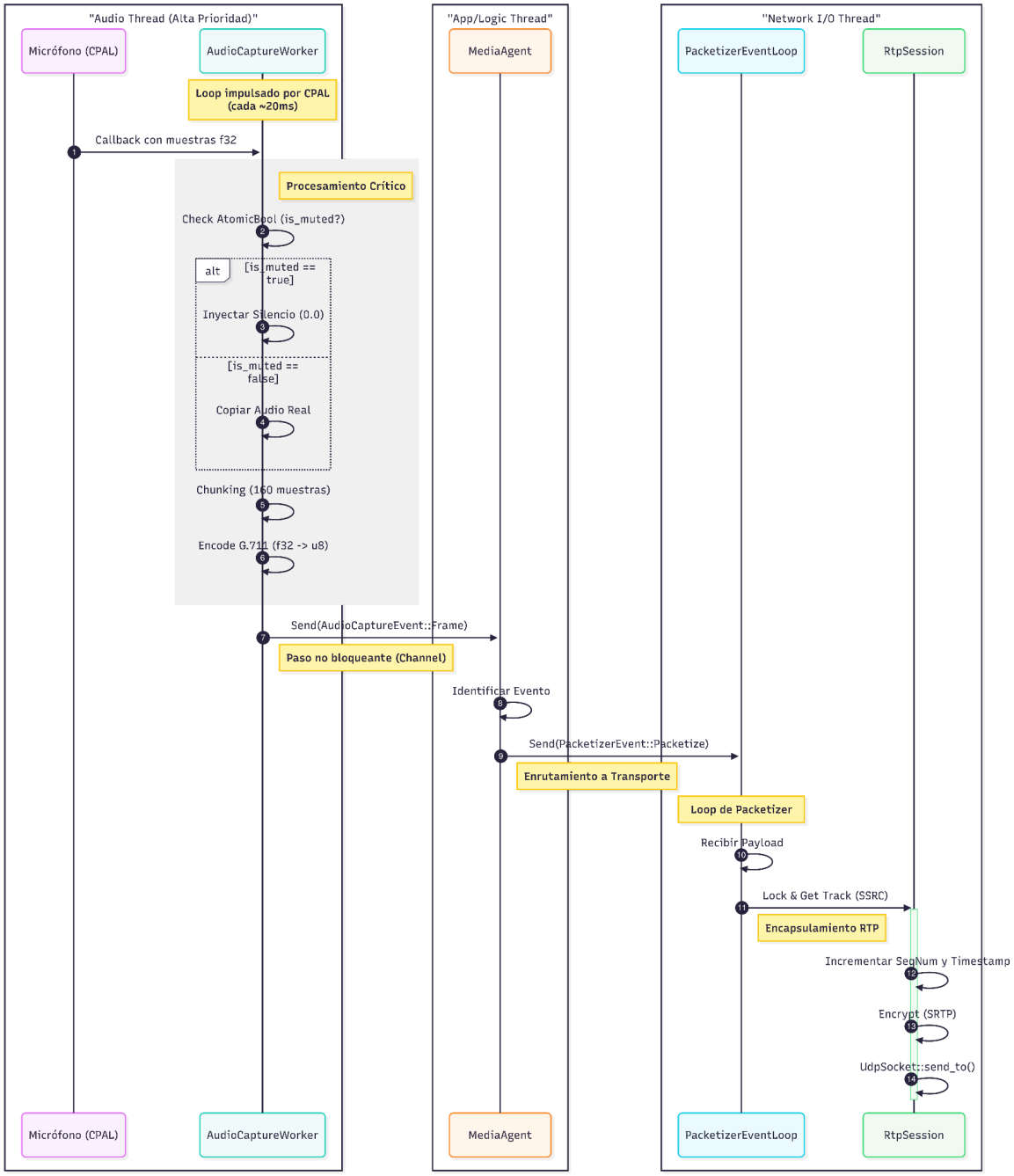

Figura 7.1 - Diagrama multi thread de la transmisión de audio

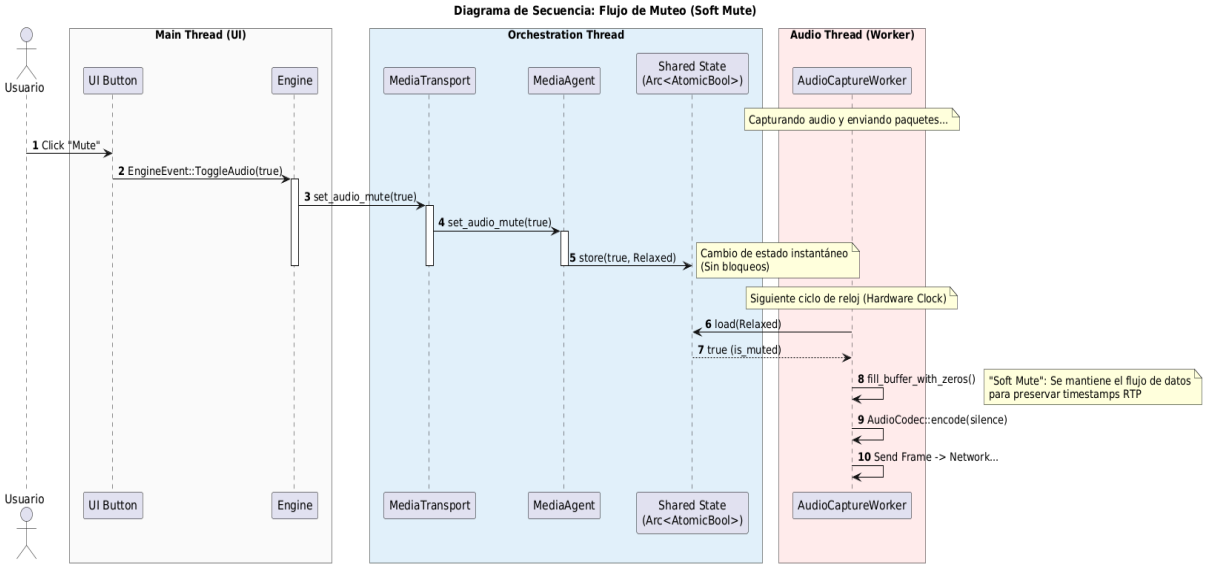

Figura 7.2 - Diagrama de secuencia para el muteo/desmuteo

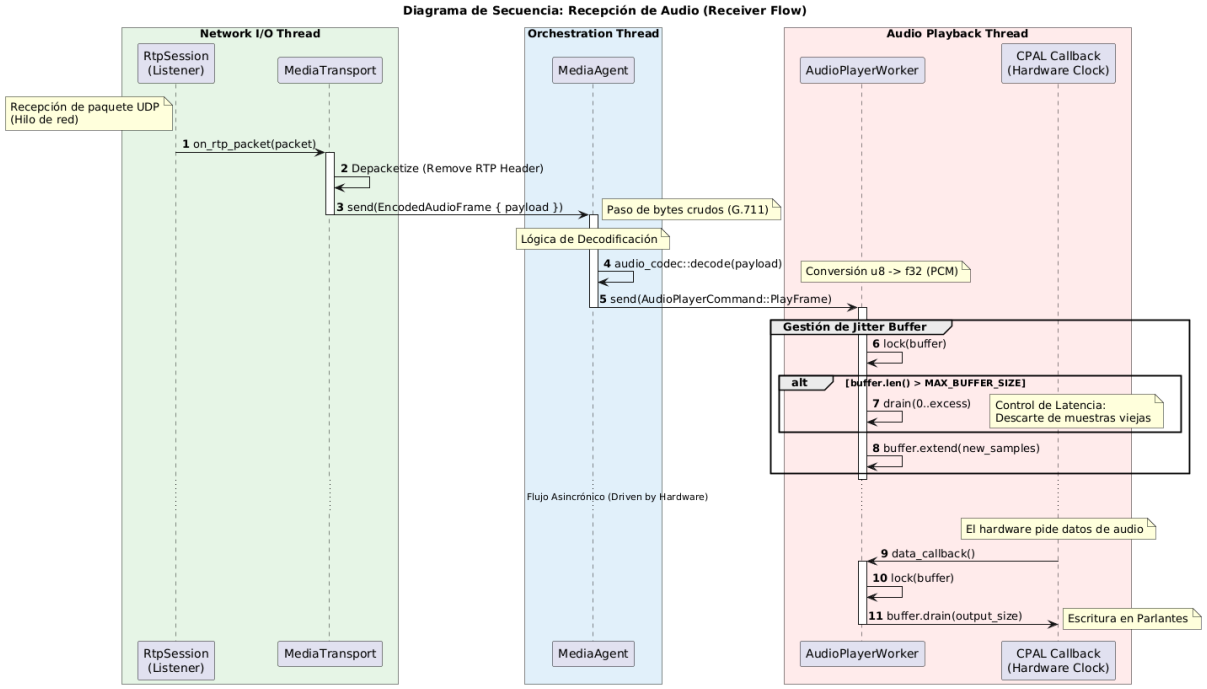

Figura 7.3 - Diagrama de secuencia para la recepción de audio

Transferencia de Archivos

Se diseñó e implementó un módulo completo para el envío y recepción de archivos de manera confiable, aprovechando el canal de datos SCTP (sobre DTLS) ya existente en la aplicación. El objetivo principal fue permitir el intercambio de archivos de cualquier tamaño sin bloquear el hilo principal de la aplicación (UI) ni interferir con la transmisión de medios en tiempo real.

Protocolo de Comunicación

Para estructurar el intercambio de datos y garantizar que los pares puedan interpretarse correctamente, se definió un protocolo de aplicación personalizado (`SctpProtocolMessage`) que opera por encima de la capa de transporte SCTP. Este protocolo añade una capa de serialización donde cada mensaje comienza con un encabezado de tipo (1 byte), seguido de los campos de datos codificados en BigEndian.

Los mensajes definidos para la máquina de estados de la transferencia son:

- **Offer (Tipo 1):** Inicia la negociación. Contiene el `transaction_id`, el tamaño del archivo y el nombre del mismo.
- **Accept (Tipo 2):** Indica que el peer remoto aceptó la solicitud y está listo para recibir datos.
- **Reject (Tipo 3) / Cancel (Tipo 4):** Permiten rechazar una oferta o cancelar una transferencia en curso.
- **Chunk (Tipo 5):** Transporta un fragmento del archivo junto con su ID y número de secuencia.
- **EndFile (Tipo 6):** Señala que se han enviado todos los fragmentos correspondientes a un ID.

Arquitectura y Estrategia Multi-threading

La implementación sigue una arquitectura orientada a eventos que desacopla las operaciones de Entrada/Salida (I/O) de disco, que son inherentemente bloqueantes, de la lógica de red y renderizado. Para ello se introdujeron los siguientes componentes y estrategias de hilos:

1. **FileHandler:** Es un actor autónomo que corre en su propio hilo (Listener). Su función es despachar comandos desde el `Engine` y gestionar el ciclo de vida de los *workers*.
2. Workers Dinámicos:
    - **ReaderWorker:** Se crea un hilo independiente para cada archivo que se envía. Su tarea es leer del disco y segmentar el archivo en *chunks*.
    - **WriterWorker:** Se crea un hilo independiente para cada archivo que se recibe. Se encarga de escribir los datos en disco y realizar la limpieza final (flush) al detectar el fin de la transmisión.
3. **Rol de la Sesión (Demultiplexación):** La `Session` es el único punto de entrada de paquetes UDP desde la red. Al recibir un datagrama, inspecciona el primer byte para determinar su naturaleza: si corresponde al rango de DTLS/SCTP (20-63), lo deriva a `SctpSession`. Es `SctpSession` quien luego desencripta y parsea el contenido para notificar eventos de alto nivel (como `ReceivedOffer` o `ReceivedChunk`) de vuelta a la `Session` y al `Engine`.

Flujo End-to-End

Envío (Sender Flow): El proceso inicia cuando el usuario selecciona un archivo. El `FileHandler` levanta un `ReaderWorker` y envía una oferta (`Offer`) al peer. Una vez aceptada la oferta, se activa un mecanismo de **Pacing (Control de Ritmo)**: el `Engine` envía periódicamente eventos `DrainChunks` al `FileHandler`. Esto dispara la lectura del siguiente fragmento en el `ReaderWorker`, evitando que la lectura de disco inunde el buffer de salida de la red.

Recepción (Receiver Flow): Los paquetes llegan a la `Session`, que los identifica como tráfico SCTP y los pasa a `SctpSession`. Una vez procesado el protocolo DTLS/SCTP, el mensaje deserializado (`Chunk`) sube como evento al `Engine`. El `Engine` reenvía el payload al `FileHandler`, indicando el `transaction_id` para que este lo enrute al `WriterWorker` específico del archivo.

Diagramas de Diseño

A continuación se presentan los diagramas de secuencia que detallan la interacción entre los hilos de lógica, transporte y I/O:

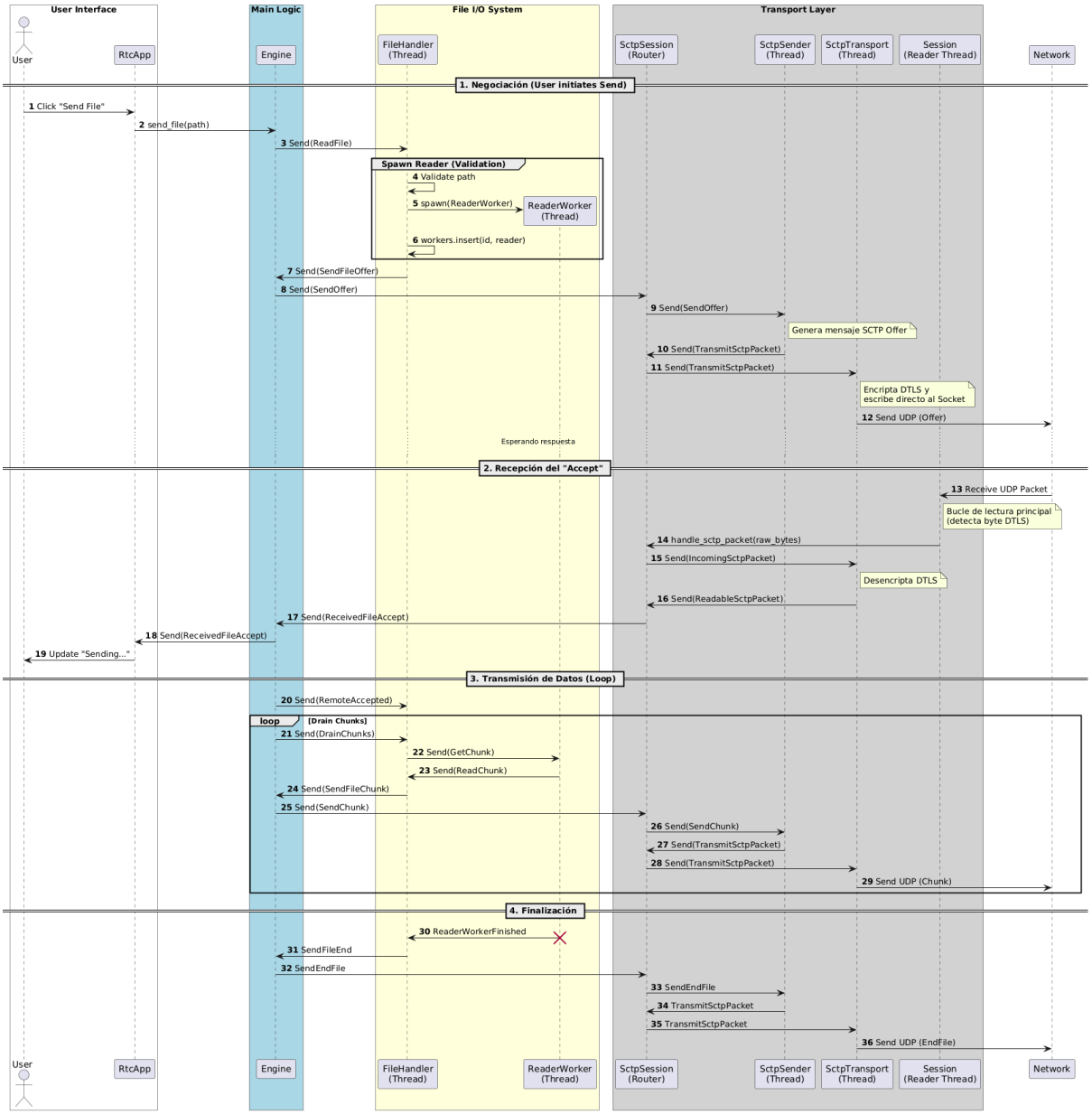

Figura 8.1 - Flujo de Envío

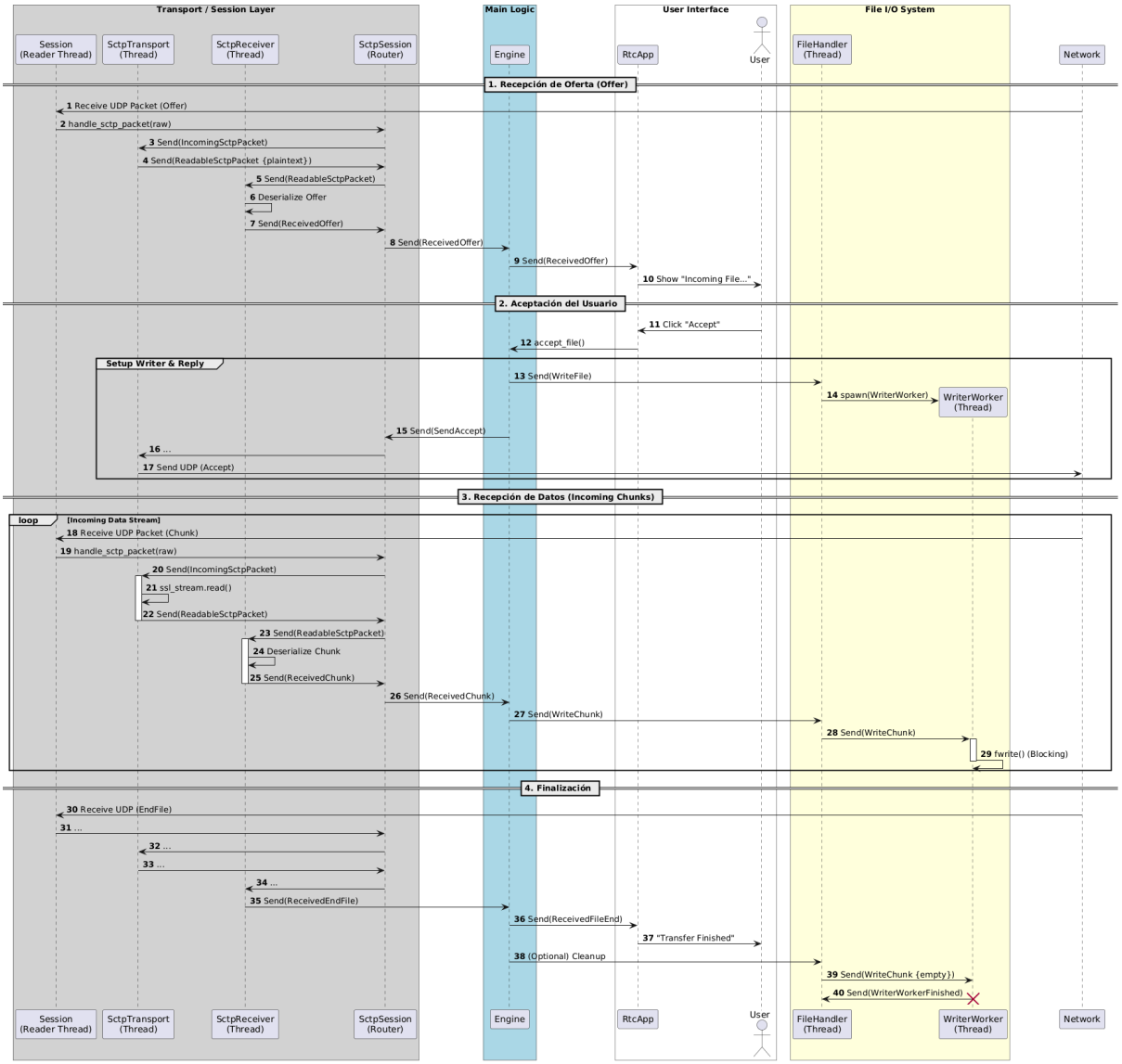

Figura 8.2 - Flujo de Recepción

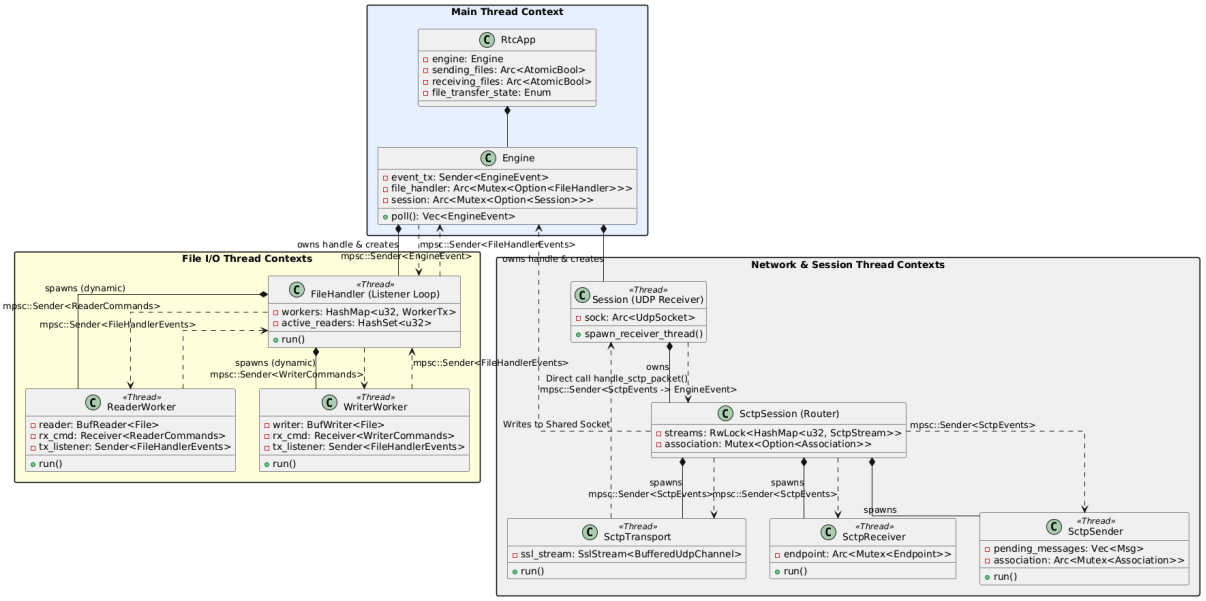

Figura 8.3 - Diagrama de Objetos de Transferencia de Archivos