FACULTAD DE INGENIERIA
Universidad de Buenos Aires

# TP: Room RTC

**[95.08] Taller de programación**
**Segundo Cuatrimestre de 2025**

## Integrantes


| Alumno | Padrón |
|---|---|
| Villa Jimenez, Alexander | 95428 |
| Pinargote Velásquez, Tom Harry | 111689 |
| Olalla Togra, Nervo Falconery | 111731 |
| Cazal Coronel, Luis Nicolás | 110104 |
| | |

# Índice

# Introducción

El presente trabajo corresponde a la entrega intermedia del proyecto de “Room RTC”, y tiene como objetivo implementar y demostrar el funcionamiento de los principales módulos del stack WebRTC, centrando en el desarrollo exclusivamente del lado del cliente y en comunicación local (sin signaling server).
La solución principalmente plantea el modelado e implementación de los componentes fundamentales que permiten establecer una conexión directa (P2P) entre dos clientes, con el objetivo de realizar una videoconferencia.

# Alcance

El alcance de la entrega comprende los siguientes puntos:

- **SDP (Session Description Protocol):** Se implementaron estructuras de datos y métodos necesarios para generar, serializar y parsear archivos SDP. Esto permite describir las capacidades de cada cliente (medios, codecs, direcciones y puertos) e intercambiar esta información para negociar la sesión de comunicación.
- **ICE (Interactive Connectivity Establishment):** Se implementó el ciclo completo del protocolo ICE, incluyendo descubrimiento de candidatos locales(gathering host), el

intercambio de los mismos, mediante SDP Offer/Answer, formación de pares, connectivity checks locales, para validar la conectividad entre pares y posteriormente la nominación y elección del par valido, para de esa manera establecer la conexión UDP.

- **Codec de video:** Se integró la librería OpenH264, para la transmisión de video, implementando un módulo llamado “Media agent”, que brinda una interfaz, para obtener la configuración del codec y para la codificación y decodificación de los frames.
- **RTP y RTCP:** Se implementaron las estructuras base de estos protocolos, permitiendo la transmisión local de video codificado, mediante el módulo “Media agent”, que utiliza la librería OpenH264, para la codificación y decodificación de frames.
- **Interfaz de usuario:** Se desarrolló una interfaz gráfica simple, que permite a dos usuarios ejecutar la conexión de forma manual. Desde esta interfaz, el Usuario A puede generar una *SDP Offer*, compartirla con el Usuario B, quien a su vez genera la *SDP Answer* y la devuelve, estableciendo así la sesión P2P y habilitando la transmisión de video.
- **Aplicación de cliente:** Es la capa superior que integra y orquesta toda la lógica que comprende todos los módulos mencionados.

Cabe aclarar que en esta instancia no se utiliza servidor de señalización, encriptación (DTLS/SRTP) ni servidores STUN/TURN, dado que el alcance se limita a conexiones locales.

# 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

En un “primer approach”, diseñamos un diagrama de arquitectura donde ejemplificamos

todo el sistema completo a alto nivel, incluyendo la relación cliente-servidor.

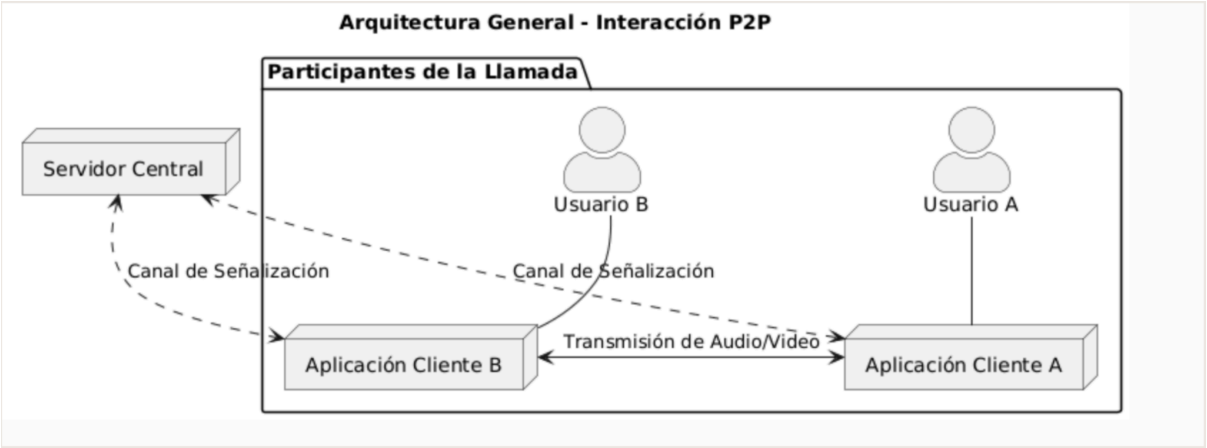



*Figura 1.1 - Arquitectura del sistema general*

Luego avanzamos en enfocarnos solamente en la arquitectura del lado del cliente, para poder ejemplificar los distintos módulos que comprenden el stack de WebRTC, y como estos se relacionan entre sí.

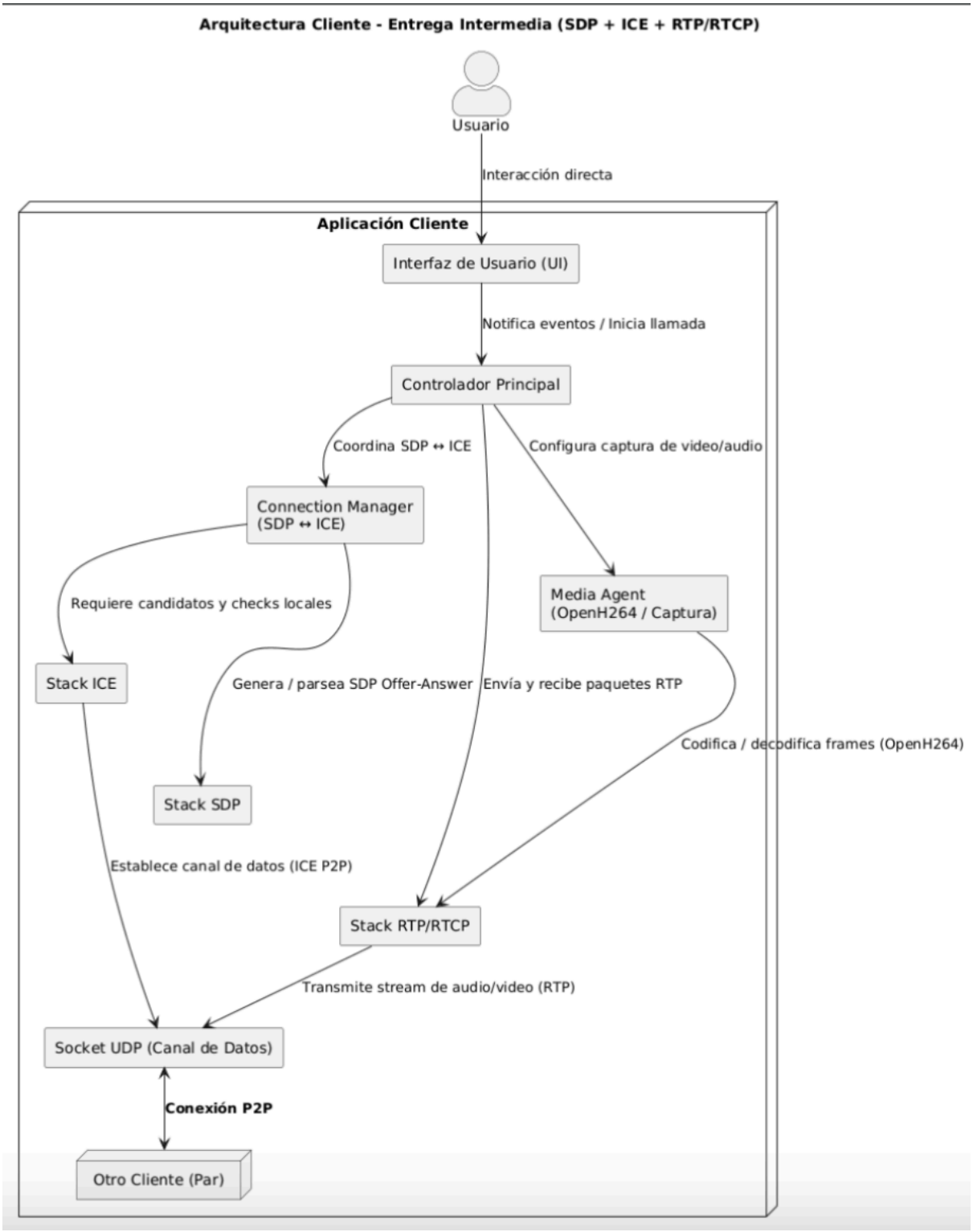



*Figura 1.2 - Arquitectura del cliente*

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

# Módulo ICE

## ¿Qué es y para qué sirve?

- ICE es un protocolo diseñado para atravesar NATs y firewalls, **especialmente para conexiones UDP**.
- En el caso de WebRTC, cuando dos clientes quieren comunicarse directamente (peer-to-peer), no basta con que cada uno informe su IP local porque muchas veces están detrás de routers/NATs. ICE ayuda a descubrir **qué direcciones / puertos** usar para alcanzar la comunicación directa.
- Para lograr esto, cada agente (cliente) recopila lo que se llama **candidatos** (candidates) de transporte (combinaciones de direcciones IP + puerto) que podrían funcionar. Luego se hacen pruebas ("connectivity checks") para ver cuáles pares de candidatos entre los dos extremos realmente pueden comunicarse.
  ICE hace uso de protocolos auxiliares como **STUN** (para descubrir la dirección pública de un host detrás de NAT) y **TURN** (para usar un servidor relay cuando no hay ruta directa posible).
- Al final, ICE (idealmente) selecciona un par de candidatos válidos que permiten que los datos fluyan entre los dos clientes, de la forma más directa y eficiente posible.

## Implementación para entrega intermedia

### Modelado de estructuras básicas

Se comenzó modelando las estructuras básicas y necesarias para poder realizar toda la etapa del protocolo. Algunas de ellas son:

- **Candidate:** Representa una dirección de red que un cliente puede ofrecer para conectarse. Contiene propiedades como una dirección(ip + puerto), el tipo(host, si es para conexión local), un socket para realizar la conexión, una prioridad(se realiza el cálculo de acuerdo a la definición del RFC), el campo "foundation" que es un identificador único que agrupa candidatos similares(también se define la lógica de acuerdo con el RFC), entre otros.
- **CandidatePair:** contiene un candidato local y remoto, una prioridad, un estado(que puede ser "InProgress", "Waiting", "Failed") y un booleano que nos dice si el par está nominado o no.
- **IceAgent:** es una de las estructuras más importantes, ya que contiene gran parte de la lógica para las diferentes etapas de la conexión. Está compuesta por un conjunto de candidatos locales, candidatos remotos, un rol("controlling" es el rol del agente que se encarga de nominar un par, "controlled" solo acepta).

### Cálculo de campo Foundation

- Según el RFC (5.1.1.3), el campo *foundation* identifica candidatos que comparten las mismas propiedades de base
- En la práctica, el foundation suele ser un **hash (string corto)** que representa la combinación de estos 3 campos. Con estos 3 campos se crea un hash, y ese valor se setea en foundation. (candidate_type, base IP, transport).

### Cálculo de campo “priority”

- Este campo se usa en el *pairing algorithm* de ICE para decidir qué candidatos son más preferidos(es decir, más aptos para establecer una conexión udp).
- Es clave al momento de realizar el “connectivity check”

Según el RFC (5.1.2.1), la fórmula es:

priority = (2^24)*(type preference) +
(2^8)*(local preference) +
(256 - component ID)

Donde:

- type preference: depende del tipo (Host, ServerReflexive, Relayed, etc.)
- local preference: puedes derivarla de la interfaz (por ejemplo, preferir IP pública sobre privada)
- component ID: normalmente 1 para RTP y 2 para RTCP.

### Rol del ICE Agent: `Controlling` vs `Controlled`

En una conexión ICE hay siempre dos agentes:

- uno asume el rol de controlling
- el otro el de controlled

Este rol no depende de quién inició la videollamada necesariamente, aunque muchas veces coincide.

### Controlling

- Es el que toma la decisión final sobre cuál par de candidatos se usará para la conexión.
- Cuando detecta que un par es válido, lo nomina.
- El otro peer debe aceptar esa nominación.

Solo uno de los dos puede nominar. Por eso este rol es único.

### Controlled

- Es el “que acepta”. Por ello, tiene un “rol“ más bien pasivo
- Responde a los connectivity checks del otro peer.
- Cuando el peer controlling, nómina un par, este simplemente lo adopta como válido.

## Proceso de “Gathering Host local”

Es la primera etapa del protocolo, donde el agente descubre las direcciones desde las cuales puede recibir conexiones directamente. Es decir, cada agente(de cada cliente) descubre sus “candidatos locales” posibles para poder establecer la conexión a futuro. Para ello, en esta fase de **“host candidate gathering”**(según el RFC), se realiza:

- Se busca las IPs locales de la máquina (interfaces activas).
- Por cada dirección encontrada, se crea un `Candidate` de tipo `CandidateType::Host`.
- Finalmente el agente setea y guarda una lista de candidatos locales

Básicamente, la máquina descubre que para salir a la red debe usar la IP local, genera un puerto dinámico, y de esa forma se conforma un “**ICE Host Candidate”** con esa dirección + puerto. Por ejemplo: `192.168.0.191:49368`

## Intercambio de pares

En esta etapa lo que se quiere lograr es que cada agente(correspondiente a cada cliente), envíe información sobre sus candidatos locales, para que de esta forma el otro “peer” pueda setear y guardar sus candidatos remotos.

En un principio habíamos optado por simular el intercambio mediante la persistencia en un archivo. Pero luego evolucionamos la funcionalidad, enviando toda la información de los candidatos locales a través de SDP. De esa forma logramos la integración entre estos 2 componentes, en un módulo intermedio llamado “Connection_manager”.

Entonces, cuando SDP envia la “oferta”, en ese momento tambien se envian los candidatos del “peer A”, el “peer B” recibe la oferta, guarda los candidatos remotos y da una respuesta,

enviando también sus candidatos locales, para que el “peer A”, realice el mismo proceso de guardado. De esta manera se completa el intercambio. En el diagrama de secuencia final, de la “Figura 3” se podrá apreciar, el flujo completo de este intercambio de candidatos.

### Ejemplo sencillo

Supongamos que tu máquina (A) tiene la IP `192.168.0.10`
y la máquina de B tiene `192.168.0.20`.

- Para **A**:
    - `192.168.0.10:xxxx` → Local candidate
    - `192.168.0.20:yyyy` → Remote candidate (viene de B, a traves de SDP)
- Para **B**:
    - `192.168.0.20:yyyy` → Local candidate
    - `192.168.0.10:xxxx` → Remote candidate (viene de A, a traves de SDP)

Entonces, **“remoto”** significa simplemente “candidato que pertenece al otro peer”.

“Local” → candidatos que tu agente generó (tus IPs y puertos, en esta caso red local).
“Remoto” → candidatos que recibió de otro agente ICE (del peer), cualquier IP/puerto que no es el tuyo. Usando sdp para el envío.

## Formación de pares y prioridades

Se inicia el proceso de combinación de candidatos locales con candidatos remotos(como si fuese un producto cartesiano), para formar un conjunto de pares(local + remoto), que se inician con un estado válido. Cabe aclarar que cuando se crea un “candidato local” también se inicializa con su respectiva “prioridad”, calculada según RFC(8445 §5.1.2.1).

Una vez que se conforma un par de candidatos, se calcula la prioridad de dicho par, combinando las prioridades individuales de los candidatos *local* y *remoto*, considerando el rol del agente (Controlling / Controlled). Segun formula en RFC 8445 §6.1.2.3

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

donde:

- **G** = prioridad del candidato del agente *controlling*
- **D** = prioridad del candidato del agente *controlled*

Cabe aclarar, que en la lógica también se realizan distintos filtros para asegurar la consistencia de los pares, al momento de conformarse. Por ejemplo, se evita formar pares con direcciones IPV4 y IPV6(ambos deben pertenecer al mismo protocolo de red), de la misma manera se evita conformar pares que tengan distintos protocolos de transporte(udp y TCP).

Por último, el agente ordena la lista de pares de candidatos de forma descendente, tal que, el par válido con mayor prioridad se encuentre al principio de la lista.

Luego de ejecutarse este proceso, el agente contiene todos los pares posibles con prioridad asignada.

### Caso borde: prioridad cero

Si se da el caso, de que un par de candidatos se origine con prioridad cero, se implementó la solución(según RFC 8445 § 5.1.2.2), de inicializarlos con un valor constante máximo “MAX_LOCAL_PREF”, el cual tiene un valor de 65535. Esto se hace para evitar inconsistencias que podría traer el hecho de tener candidatos con prioridad cero. También, se tomó un valor máximo para formar pares candidatos “MAX_PAIR_LIMIT”(con valor 100, según RFC).

## Connectivity Checks

En esta etapa la idea es simular el comportamiento de los *Binding Requests* (usados en STUN), pero de forma local. Es decir, probar conectividad local UDP entre candidatos pares (CandidatePair) ya formados.
Esto permitirá detectar si los sockets locales pueden intercambiar mensajes en red local. Es una especie de testeo previo de la conexión. Una vez que se envía el mensaje, se le cambia el estado al “par candidato” y está listo para ser elegido en el próximo paso.

### Pasos de connectivity Checks

- Se inicia el proceso(se itera la lista) para todos los “pares de candidatos” que hay en el agente, y que se encuentran en estado “Waiting”.
- Se verifica si el socket del candidato local es válido, sino se deja al par en estado “Failed”.
- El candidato local envía un “binding request", sin esperar respuesta, al candidato remoto. BINDING REQUEST es un mensaje string constante, que toma valor de "BINDING-REQUEST", el cual tomamos de ejemplos investigados en internet.
- Si todo sale bien, se cambia el estado a “InProgress”.
- Cuando se recibe la respuesta del candidato remoto, el estado del par pasa a “Succeeded”.

Al terminar el proceso, se tiene una lista de candidatos con estado exitoso y algunos en estado fallido. La instancia de nominación, sólo tomará en cuenta a los pares que hayan finalizado el proceso de “connectivity check” en estado exitoso.

## Selección de par válido y nominación

En esta etapa se selecciona el par válido(nominación), según el rol del agente y la prioridad.

### Pasos:

- De toda la lista de pares de candidatos que se encuentran en el agente, nos quedamos solamente con los que tienen estado “Succeeded”.
- Si no hay pares en estado “Succeeded” se sale por error.
- Me quedo con el par de mayor prioridad.
- Si el agente tiene el rol de “controlling”, se marca el par como “nominado”. El rol de “controlled” solo acepta la nominación.

### Rol Controlling

El Controlling Agent, toma decisiones activas, es decir, elige cuál “par” usar para establecer la comunicación final (simulando el comportamiento del flag USE-CANDIDATE en STUN, cuando se implemente el servidor).

### Rol Controlled

Este agente no decide, simplemente espera que el “controlling” nomine un par.

## Establecimiento del canal de datos(UDP)

Una vez que ya se acepta el par válido nominado, el paso final es abrir el socket UDP, local y remoto, para establecer la conexión “p2p” y comenzar enviar los paquetes de datos(frames), para la transmisión de vídeo.

### Pasos detallados:

- Se valida la existencia del par nominado, sino se sale por error.
- Se abre el socket local, del par candidato.
- Se envía un mensaje de prueba al candidato remoto. Por ejemplo, un mensaje “Hola ICE”.
- Se espera el “acknowledge” de respuesta, sino se sale por error.
- Una vez recibida la respuesta del candidato remoto, queda establecida la conexión UDP, y se puede comenzar a transmitir los frames del video.

## Diagrama de secuencia del flujo completo de ICE

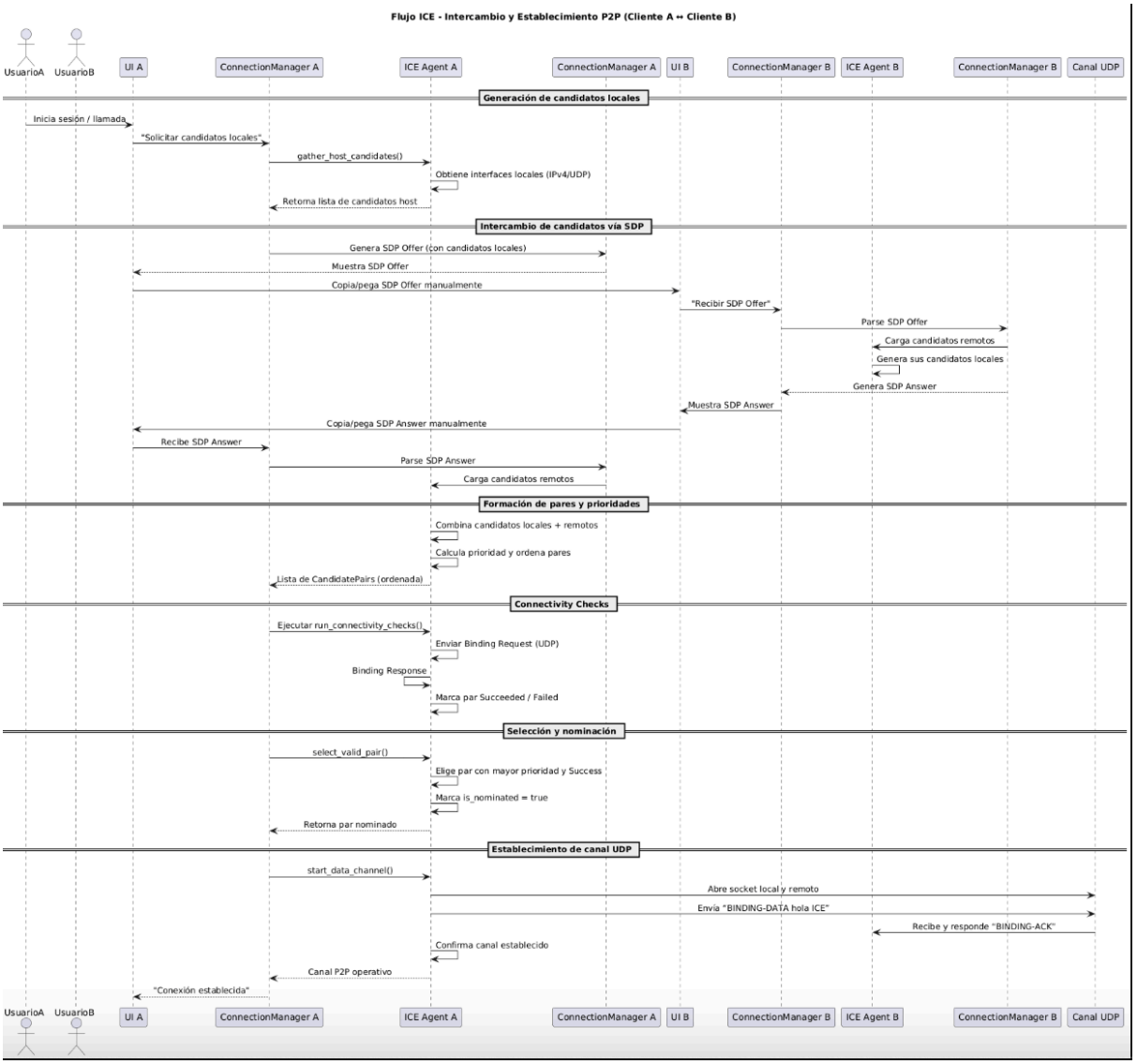



*Figura 1.3 - Flujo completo de ICE*

# Módulo SDP

## ¿Qué es y para qué sirve?

**SDP (Session Description Protocol)** es un protocolo de formato de texto cuyo propósito es *describir* una sesión de comunicación multimedia. No transporta ningún dato de audio o video por sí mismo; su única función es actuar como una "tarjeta de presentación" o un "anuncio clasificado" que un participante le envía a otro para negociar *cómo* será la llamada.

En el contexto de nuestro proyecto, el SDP responde a dos preguntas críticas:

1. **¿Qué enviaremos?** (Negociación de Medios): "Propongo que usemos el códec de

video H264 (que llamaré 'tipo 96') y que el video fluya por el puerto 9".
2. **¿Por dónde nos conectamos?** (Integración con ICE): "Para que puedas encontrarme, aquí están mis 'candidatos' de red (direcciones IP y puertos) que mi agente ICE descubrió".

El objetivo de nuestro módulo `sdp` es modelar este formato de texto en estructuras de Rust, permitiéndonos **parsear** (leer) ofertas SDP entrantes y **generar** (escribir) nuestras propias ofertas y respuestas.

## Implementación para entrega intermedia

Para la entrega intermedia, la implementación del "Stack SDP" (como se ve en el diagrama de arquitectura) se centró en tres pilares:

- **Modelado de Datos:** Crear representaciones internas que reflejen la naturaleza jerárquica de un documento SDP.
- **Parseo (Deserialización):** Implementar la lógica para convertir un texto SDP en nuestras estructuras internas.
- **Generación (Serialización):** Implementar la lógica para convertir nuestras estructuras internas de nuevo en un texto SDP con formato válido.

## Modelado de estructuras básicas

A diferencia de un formato simple, SDP tiene una jerarquía. Hay campos que aplican a toda la sesión (globales) y campos que aplican solo a un flujo de medios (ej: solo al video). Nuestro modelado de datos refleja esto:

- **Atributo:** Es la unidad más fundamental. Modela una línea de tipo `a=` (ej: `a=rtpmap:96 H264/90000`) como una simple estructura de clave y valor.
- **Descripción de Medio:** Esta es la estructura clave que modela una sección de medios, la cual siempre empieza con una línea `m=` (ej: `m=video...`). Su propósito es agrupar la información de un **único** flujo (como el video). Por lo tanto, contiene el tipo de medio (video, audio) , los formatos que acepta y, crucialmente, su **propia lista de atributos** que aplican solo a este medio (como los códecs o los candidatos de ICE).
- **Documento SDP:** Es la estructura raíz que representa el documento entero. Contiene campos globales (a nivel de sesión, como el origen y el nombre) , una lista de atributos **globales** , y una lista de todas las **descripciones de medios** (ej: una para video, otra para audio).

Este diseño de composición (un `Documento SDP` contiene `Descripciones de Medios`, que a su vez contienen `Atributos`) nos permite modelar con precisión la jerarquía del protocolo.

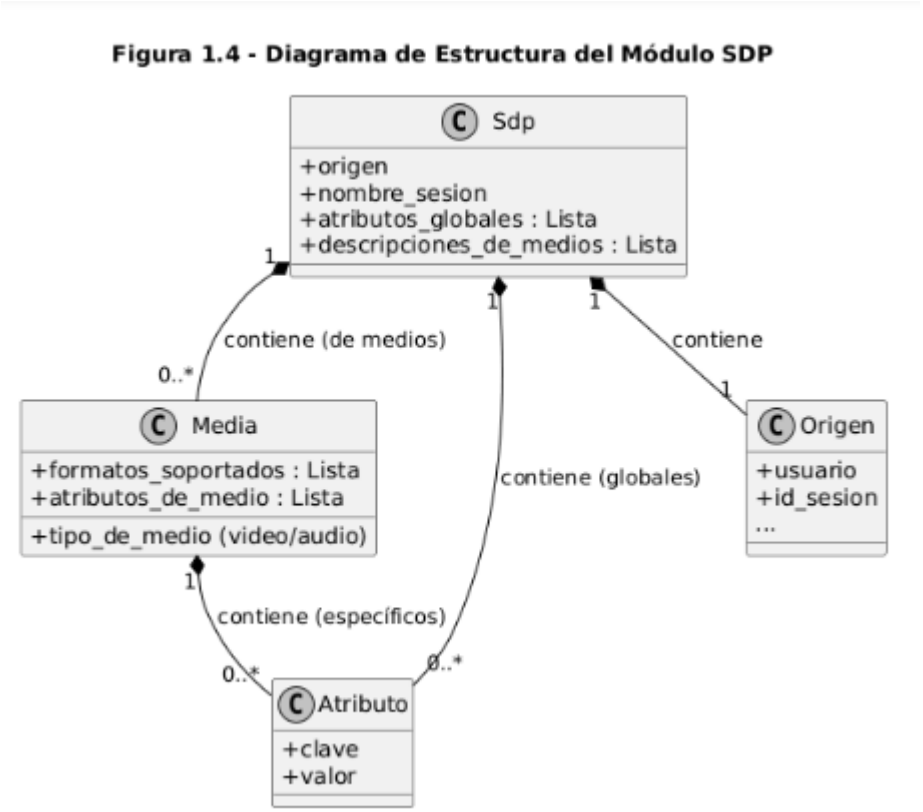



*Figura 1.4 - Diagrama de Estructura del Módulo SDP*

## Proceso de Parseo (Deserialización)

La conversión de texto a nuestras estructuras internas es manejada por un parser que funciona como una máquina de estados simple.

1. Itera por el texto de entrada, dividiéndolo por saltos de línea.
2. Para cada línea, extrae el prefijo (ej: `v=`, `o=`, `m=`, `a=`).
3. **Lógica clave:** El parser mantiene un **estado interno** para saber si se encuentra dentro de una sección de medios (es decir, si ya ha visto una línea `m=`).
4. Al inicio, este estado está inactivo. Todos los atributos (`a=`) que se leen se añaden a la lista **global** del documento SDP.
5. Cuando el parser encuentra una línea `m=`, crea una nueva estructura de `Descripción de Medio` y activa el estado.
6. A partir de ese punto, todos los atributos (`a=`) siguientes se añaden **dentro** de esa `Descripción de Medio`.

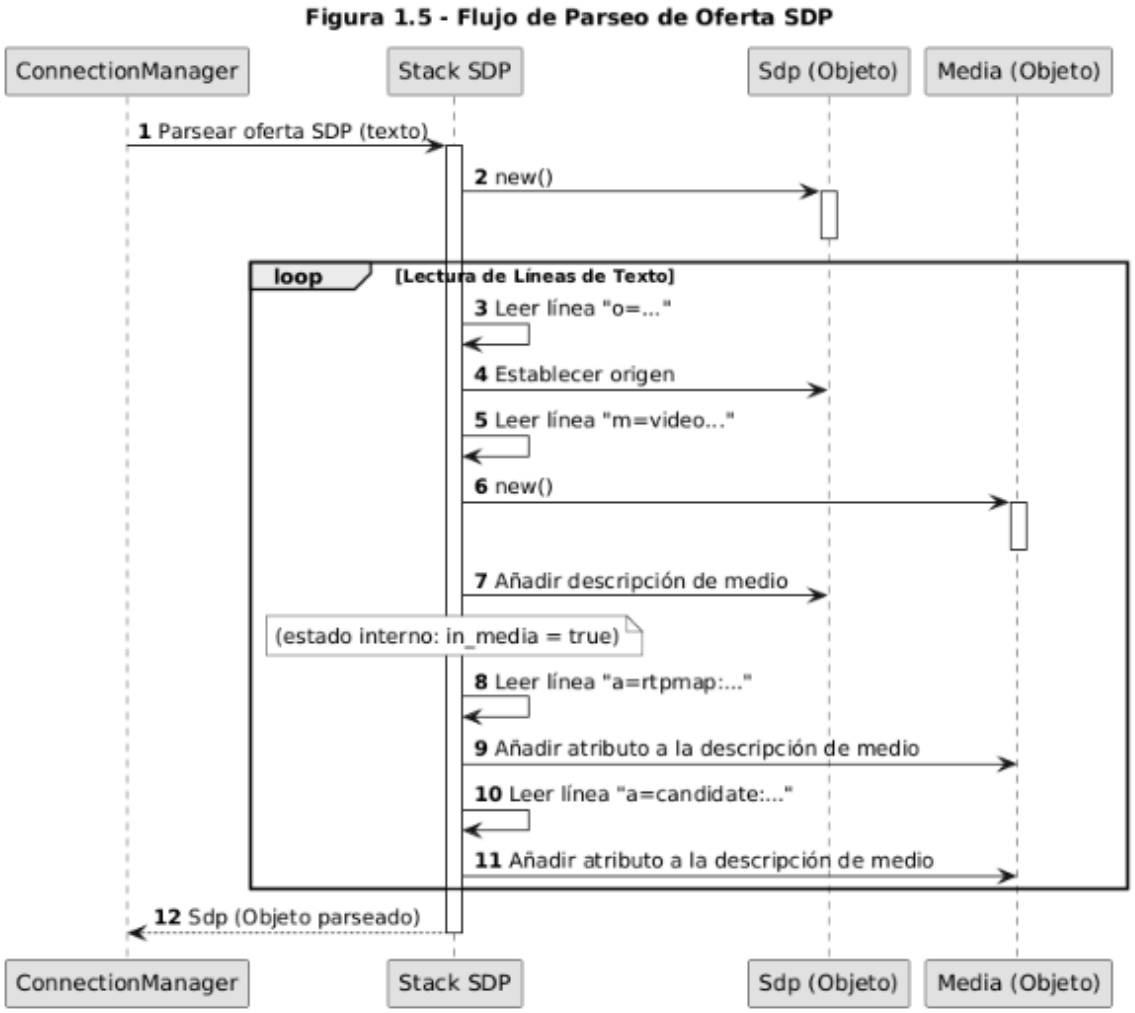



*Figura 1.5 - Diagrama de Secuencia del Flujo de Parseo de Oferta*

# Proceso de Generación (Serialización)

La conversión de nuestras estructuras internas a texto funciona a la inversa, demostrando la encapsulación de nuestro diseño.

1. Se escriben los campos de nivel de sesión (globales) en un string (ej: `v=`, `o=`, `s=`).
2. Se itera sobre la lista de atributos **globales** y se escribe cada uno.
3. **Lógica clave:** Se itera sobre la lista de **Descripciones de Medios**. Por cada una, se **delega** la responsabilidad de serialización a esa estructura.
4. La propia `Descripción de Medio` se encarga de imprimir su línea `m=` y luego iterar sobre su lista interna de atributos para imprimir todos sus atributos específicos (`a=...`).

Este diseño asegura que cada componente es responsable de serializarse a sí mismo, manteniendo el código limpio y modular.

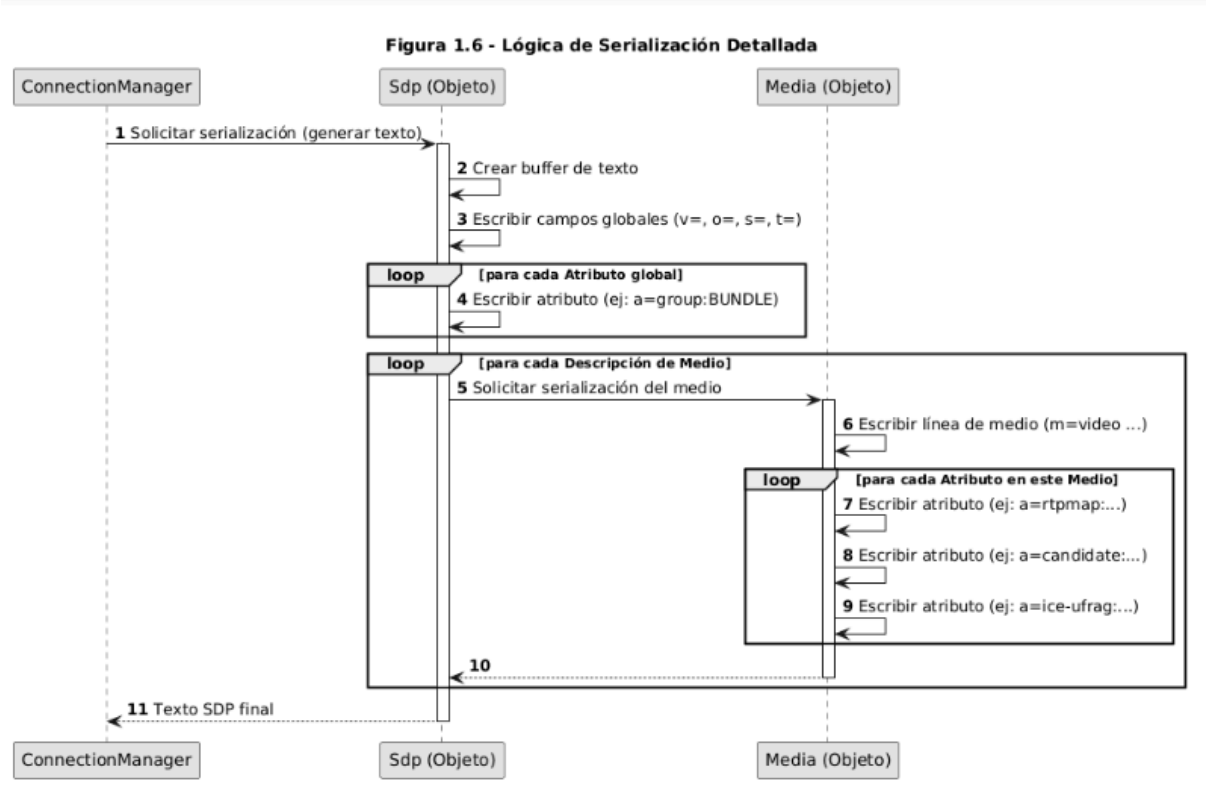



*Figura 1.6 - Diagrama de Secuencia (Lógica de Serialización Detallada)*

# Cómo se integran los módulos ICE y SDP

Como se muestra en el diagrama de arquitectura del cliente (Figura 1.2), los módulos "Stack ICE" y "Stack SDP" son componentes de lógica pura que no interactúan directamente entre sí.

La **integración** y **orquestación** entre ambos es la responsabilidad principal del módulo **ConnectionManager**.

Esta relación se puede resumir de la siguiente manera:

- El **ConnectionManager** actúa como el **orquestador**.
- El **"Stack SDP"** actúa como una librería de **serialización/deserialización**. El orquestador le pide que convierta texto en objetos (parseo) o que convierta objetos en texto (generación).
- El **"Stack ICE"** actúa como un servicio de **descubrimiento y validación de red**. El orquestador le pide que descubra candidatos locales y le entrega candidatos remotos cuando los recibe.
- Un módulo **"traductor"** específico se encarga de convertir la representación de un `Candidato` de ICE en un atributo de texto SDP (y viceversa).

## Flujo Integrado de Generación (Serialización)

El primer flujo de integración ocurre cuando el cliente (Usuario A) genera una Oferta SDP. El `ConnectionManager` debe consultar al "Stack ICE" para descubrir los candidatos locales e inyectarlos en el documento SDP antes de serializarlo.

El flujo es el siguiente:

1. El `ConnectionManager` determina que necesita crear una Oferta.
2. **Integración:** Primero, invoca al "Stack ICE" para obtener la lista de **candidatos locales**.
3. Paralelamente, define los códecs locales (H264).
4. Luego, usa el "traductor" para formatear tanto los candidatos como los códecs como **atributos SDP** (`a=candidate...` y `a=rtpmap...`).
5. Construye el objeto `Sdp` (en memoria) completo, insertando estos atributos en la `Descripción de Medio` correspondiente.
6. Finalmente, pasa este objeto completo al "Stack SDP" y llama a la función de **serialización** (`encode`) para obtener el texto final que el usuario copia y pega.

**Figura 1.7 - Flujo de Generación de Oferta**

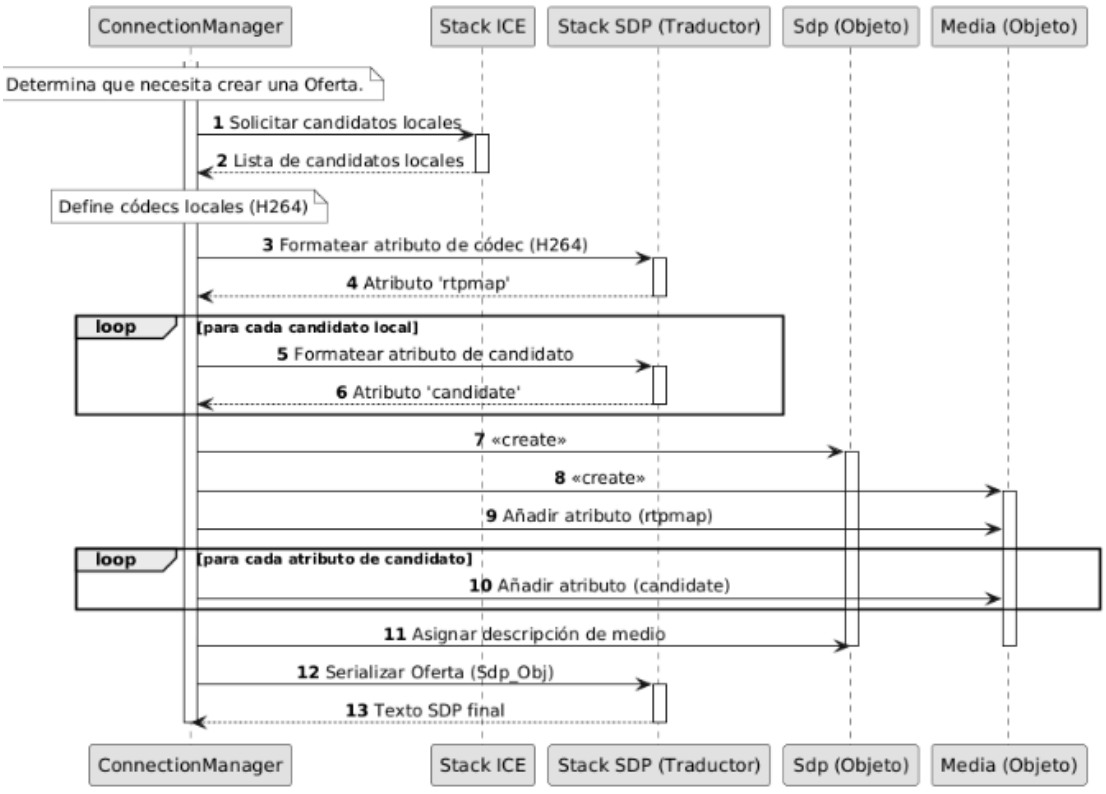



*Figura 1.7 - Diagrama de Secuencia (Flujo de Generación de Oferta)*

# Flujo Integrado de Aplicación (Deserialización)

El flujo opuesto ocurre cuando el cliente (Usuario B) recibe la Oferta de A. El `ConnectionManager` debe desempaquetar la información, entregando los candidatos al "Stack ICE" y los códecs al "Media Agent".

El flujo es el siguiente:

1. El `ConnectionManager` recibe el texto de la Oferta.
2. Invoca al "Stack SDP" para **parsear** el texto, convirtiéndolo en un objeto `Sdp` en memoria.

3. Una vez que tiene el objeto, el `ConnectionManager` lo **inspecciona** y llama a funciones auxiliares para extraer la información.
4. **Integración inversa:** Itera sobre todos los atributos de las secciones de medios.
5. Cuando encuentra un atributo de candidato (`a=candidate...`) , utiliza el "traductor" para parsear su valor y convertirlo en un objeto `Candidato` (entendible por el "Stack ICE").
6. Inmediatamente, **entrega** este objeto `Candidato` al "Stack ICE" (específicamente al `IceAgent`), que lo almacena como un candidato remoto.
7. De esta forma, el `ConnectionManager` utiliza el "Stack SDP" como parser y luego **distribuye** la información extraída a los módulos correspondientes, preparando al `IceAgent` para que inicie los *connectivity checks*.

El siguiente diagrama de secuencia detalla este flujo de orquestación para parsear una Oferta.

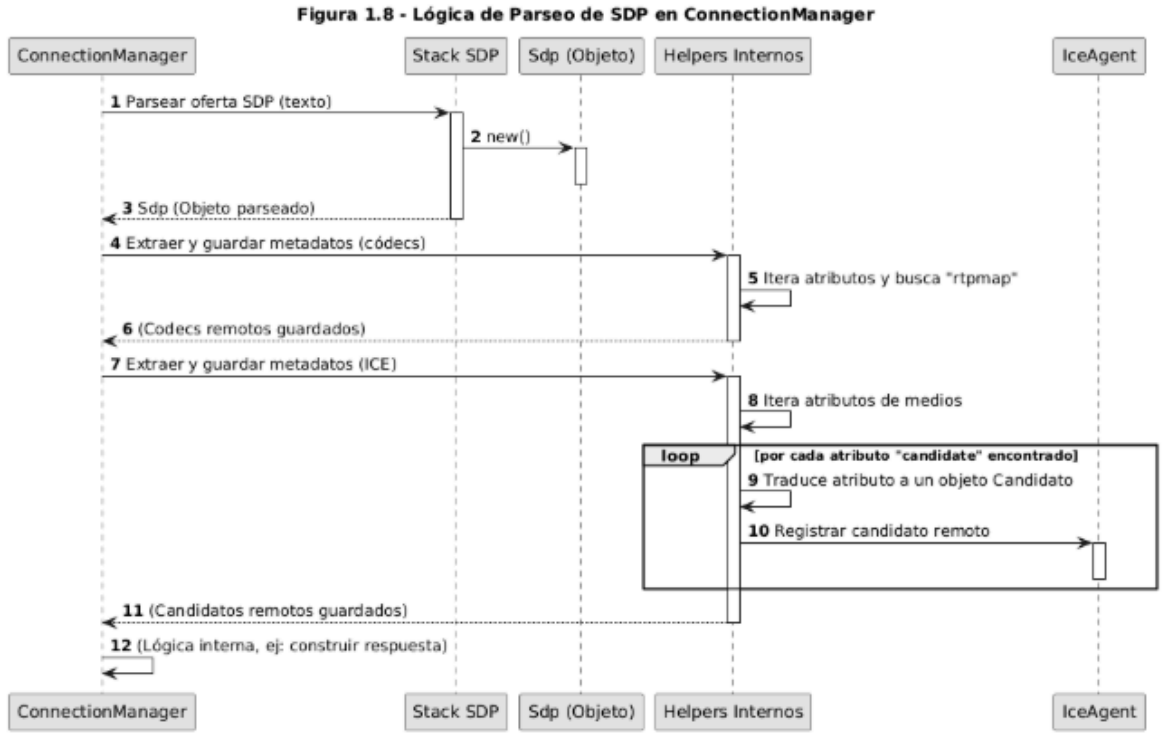



*Figura 1.8 - Diagrama de Secuencia (Logica de parseo de SDP en ConnectionManager)*

# Codec openH264

Al momento de elegir el codec a utilizar para la transmisión de video, optamos por varias opciones.
Inicialmente evaluamos el uso del códec H.264 mediante el binding x264, dado que es el más utilizado en implementaciones profesionales de WebRTC.

Sin embargo, debido a la falta de mantenimiento del crate y a problemas de compatibilidad con Rust estable, optamos por utilizar OpenH264, la implementación open source de Cisco, que ofrece un binding moderno y estable en crates.io.

Esta elección nos permitió una integración más fluida y multiplataforma, manteniendo compatibilidad con el estándar H.264 y facilitando la demostración local del códec en Room RTC.

# ¿Qué es OpenH264?

Es una biblioteca de códecs de código abierto creada por Cisco que permite la codificación y decodificación de video H.264 en tiempo real, por ej en en formato H.264/MPEG-4 AVC. Por ejemplo es especialmente útil en aplicaciones que requieren procesamiento de video en tiempo real(WebRTC).

# ¿Cómo funciona?

Es una implementación ligera y práctica del códec H.264.
A diferencia de x264, está implementado en c++(x264 está en C), y además su integración en Rust es directa, estable, portable y cumple con el estándar WebRTC.

Algunas ventajas y desventajas que analizamos y nos ayudaron a justificar nuestra decisión de usar este codec en específico, son:

## Ventajas

- **Compatibilidad:** con WebRTC y SDP, según el RFC 6184, además que se puede integrar con ICE.
- **Implementación liviana y portable:** no requiere configuraciones complejas, y aporta una compilación llevadera.
- **Mantenibilidad:** Se cuenta con un crate actualizado en crates.io (openh264 = "0.3"), estable y compatible con Rust 2021/2024.
- **Uso en WebRTC real:** Cisco Webex y Firefox lo utilizan internamente, garantizando interoperabilidad con otros clientes.

## Desventajas

- **Rendimiento:** tiene un peor rendimiento, en entornos de alta exigencia, en comparación con x264, puede tener menor eficiencia en CPU o calidad visual.
- **Sin soporte de audio integrado:** Solo cubre el códec de vídeo; el códec de audio debe integrarse aparte (Opus, AAC, etc.).
- **Escasa documentación:** Menos documentación detallada en Rust, en comparación con x264(de H.264), el crate tiene menos ejemplos oficiales.

Un aspecto importante que nos gustaría volver a remarcar es que, el hecho de no incluir soporte para audio, no sería un inconveniente, ya que a futuro sólo se debería añadir algún códec de audio como Opus y se deberá realizar la extensión correspondiente con SDP.

# Protocolos RTP y RTCP

## ¿Qué es y para qué sirve?

### RTP (Real Time Transport Protocol)

RTP es un protocolo de capa de aplicación diseñado para la entrega de datos en tiempo real, como audio y video, a través de redes IP. En nuestro proyecto, la implementación de RTP se centra en:

- **Transporte de medios:** Encapsular los datos de video y futuramente de audio (payload) en paquetes RTP para su transmisión.
- **Sincronización:** Utilizar números de secuencia y marcas de tiempo para permitir al receptor detectar la pérdida de paquetes, reordenarlos y reconstruir el flujo de medios de manera sincronizada.
- **Identificación de fuentes:** Asignar identificadores SSRC (Synchronization Source) y CSRC (Contributing Source) para distinguir las diferentes fuentes de medios en una sesión.
- **Identificación de codec:** Indicar el tipo de codec utilizado para la carga útil (payload type), permitiendo al receptor decodificar correctamente los datos.

### RTCP (RTP Control Protocol)

RTCP es un protocolo complementario a RTP que proporciona funcionalidades de control y retroalimentación para las sesiones de medios en tiempo real. Su propósito principal en nuestro proyecto es:

- **Control de Calidad de Servicio:** Enviar informes periódicos sobre la calidad de la transmisión (pérdida de paquetes, jitter, RTT) para que los participantes puedan ajustar sus parámetros de envío.
- **Identificación de participantes:** Distribuir información sobre los participantes de la sesión, como el CNAME (Canonical Name).
- **Manejo de Eventos:** Permitir la señalización de eventos importantes, como solicitudes de keyframe (PLI) o solicitudes de retransmisión (NACK).

## Modelado de estructuras básicas de RTP

- `struct RtpPacket` (`rtp_packet.rs`)**:** Representa un paquete RTP completo, incluyendo su cabecera y la carga útil (payload).
- `struct RtpHeader` (`rtp_header.rs`)**:** Representa un paquete RTP completo, incluyendo su cabecera y la carga útil (payload).
- `struct RtpHeaderExtension` (`rtp_header_extension.rs`)**:** Estructura para manejar las extensiones de cabecera RTP, permitiendo añadir metadatos adicionales a los paquetes RTP.

## Modelado de estructuras básicas de RTCP

- `enum RtcpPacket` (`rtcp.rs`)**:** Un enum que encapsula los diferentes tipos de paquetes RTCP soportados por la implementación. Esto permite un manejo

unificado de los distintos mensajes de control.
- **`struct CommonHeader` (`common_header.rs`):** La cabecera común presente en todos los paquetes RTCP, que incluye la versión, el bit de padding, el contador de recepción/formato y el tipo de paquete.
- **`struct RtpHeaderExtension` (`rtp_header_extension.rs`):** Estructura para manejar las extensiones de cabecera RTP, permitiendo añadir metadatos adicionales a los paquetes RTP.
- **Paquetes específicos:** El módulo rtcp contiene estructuras dedicadas para cada tipo de paquete RTCP, como `SenderReport`, `ReceiverReport`, `Sdes`, `Bye`, `App`, `GenericNack` y `PictureLossIndication`, cada una con los campos específicos definidos por el estándar.

# Integración de RTP y RTCP

La integración de RTP y RTCP en este proyecto se implementa principalmente a través del módulo `rtp_session`, que actúa como el orquestador central para el transporte de medios. Este módulo tiene la responsabilidad de mantener el estado de las transmisiones y recepciones de medios, gestionar instancias de `RtpRecvStream` y `RtpSendStream`, procesar los paquetes UDP entrantes y asignar los paquetes RTP a sus respectivas instancias de recepción basándose en su SSRC o tipo de carga útil (PT). Además, maneja los paquetes RTCP entrantes, genera y envía periódicamente paquetes RTCP, y administra el envío de medios RTP hacia los destinos correspondientes.

En primer lugar, la **gestión de flujos de medios** recae en la estructura `RtpSession`, encargada de crear y administrar las instancias de `RtpRecvStream` para los medios entrantes y de `RtpSendStream` para los medios salientes. Los `RtpRecvStream` pueden encontrarse en un estado “pendiente” (`pending_recv`) cuando aún no se ha identificado su SSRC remoto, lo que permite emparejar dinámicamente los flujos RTP entrantes conforme se detectan.

El **procesamiento de paquetes UDP** comienza cuando el método `start()` de `RtpSession` inicia dos hilos principales: uno dedicado a la recepción de paquetes UDP y otro al envío periódico de paquetes RTCP. El hilo receptor clasifica los paquetes entrantes como RTP o RTCP utilizando la función `is_rtcp`, garantizando que cada uno se procese en su contexto adecuado.

En cuanto al **manejo de RTP entrante**, los paquetes se decodifican mediante `RtpPacket::decode()`. Tras la decodificación, se redirigen al `RtpRecvStream` correspondiente en función de su SSRC. Si el SSRC no se reconoce, `RtpSession` intenta asociarlo con un flujo pendiente que coincida con el tipo de carga útil, lo que facilita el descubrimiento dinámico de nuevas fuentes RTP.

Por su parte, el **manejo de RTCP entrante** se realiza a través de la decodificación de paquetes compuestos con `RtcpPacket::decode_compound()`. Una vez decodificados, `RtpSession` procesa los diferentes tipos de mensajes RTCP:

- Los **Sender Reports (SR)** actualizan el estado de los `RtpRecvStream` (para la sincronización del reloj) y de los `RtpSendStream` (para el cálculo del RTT a partir de los bloques de informe).

- Los **Receiver Reports (RR)** proporcionan retroalimentación sobre la calidad de la recepción, informando pérdidas de paquetes y jitter.

- Los **Picture Loss Indication (PLI)** notifican la necesidad de un keyframe, lo cual se registra y podría reenviarse a un agente de medios para su generación.

- Los **Negative Acknowledgement (NACK)** indican la pérdida de paquetes específicos y solicitan su retransmisión, acción que sería gestionada por el `RtpSendStream` correspondiente.

- Finalmente, los **Goodbye (BYE)** notifican que una fuente ha abandonado la sesión, lo que provoca la eliminación de los `RtpRecvStream` asociados.

El **envío periódico de RTCP** es manejado por un hilo independiente encargado de construir y transmitir paquetes RTCP compuestos hacia el par remoto. Este hilo genera `SenderReport`s a partir de los `RtpSendStream` activos, agrega `ReportBlock`s de todos los `RtpRecvStream` para construir los `ReceiverReport`s y envía paquetes `SDES` que contienen el CNAME local, utilizado para la identificación de la fuente. Además, el sistema puede emitir `PLI`s explícitos a través del método `send_pli` para solicitar un keyframe a un remitente remoto.

Por último, el **envío de medios RTP** se realiza mediante los métodos `send_frame`, `send_rtp_payload`, `send_rtp_payloads_for_frame` y `send_rtp_chunks_for_frame`, expuestos por `RtpSession`. Estos métodos permiten que otras partes de la aplicación transmitan datos de medios y delegan en los `RtpSendStream` la construcción y envío de los paquetes RTP, encargándose de la numeración de secuencias, la gestión de marcas de tiempo y la correcta encapsulación de los datos antes de su transmisión.

## Diagrama de Clases

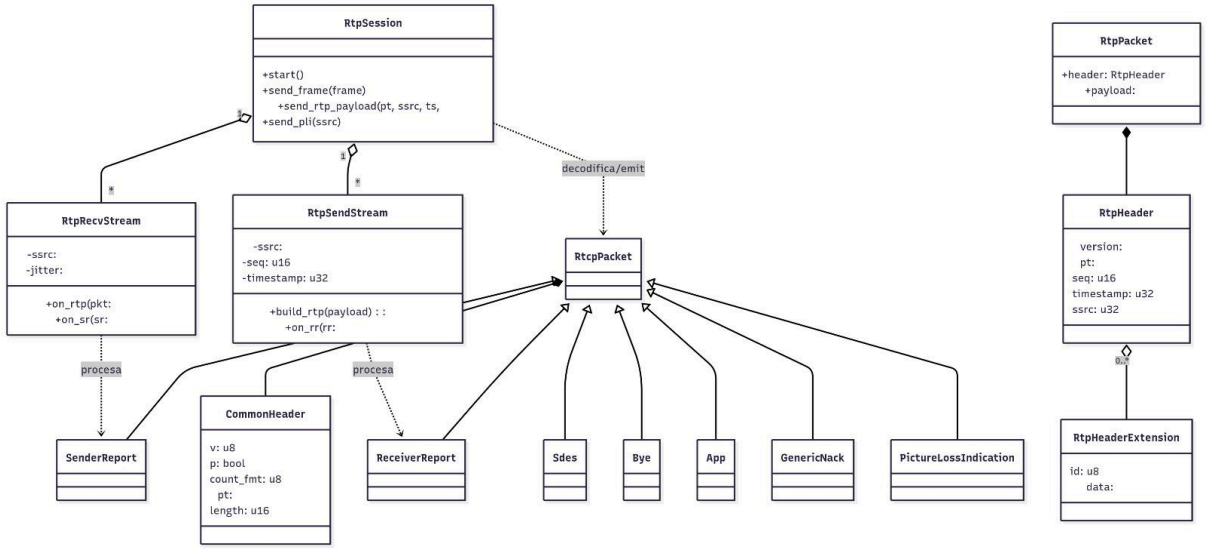



*Figura 1.9 - Diagrama de Clases (Relación de RTP/RTPC con RtpSession)*

# MediaAgent: Procesamiento de medios

El módulo `media_agent` constituye el componente central de procesamiento de medios en el proyecto nuestro proyecto. Su propósito es gestionar el flujo completo de video, desde la captura y codificación hasta la decodificación y visualización, actuando como el punto de conexión entre las fuentes de video locales (como una cámara) y el sistema de transporte de red, gestionado por `RtpSession`. En esencia, `media_agent` se encarga de facilitar el intercambio de video en tiempo real entre pares dentro del ecosistema de nuestro proyecto.

## Funciones principales

- **Gestión de Entrada de Video:** Captura fotogramas de video de la cámara local (o genera un patrón de prueba si no hay cámara).
- **Codificación de Video:** Transforma los fotogramas de video crudos en un formato comprimido (H.264) usando el crate `OpenH264` adecuado para la transmisión por red.
- **Empaquetado RTP:** Divide los datos de video codificados en unidades más pequeñas (chunks) que pueden ser transportadas dentro de paquetes RTP.
- **Desempaquetado RTP:** Reconstruye los datos de video codificados a partir de los chunks RTP recibidos.
- **Decodificación de Video:** Convierte los datos de vídeo comprimidos recibidos de nuevo a fotogramas de video crudos para su visualización.
- **Manejo de Eventos:** Responde a eventos del motor de la aplicación, como el establecimiento de una sesión o la llegada de paquetes RTP.
- **Provisión de Fotogramas** Ofrece la capacidad de obtener los últimos fotogramas de video local y remoto para su visualización en la interfaz de usuario.

## Modelado de MediaAgent

- **`struct MediaAgent` (`media_agent.rs`)**: La estructura fundamental del módulo, coordina todas las operaciones de medios. Contiene canales de comunicación (`event_tx` y `rtp_tx`) para interactuar con el motor principal y el hilo de decodificación de RTP, un manejador del hilo de decodificación RTP (`rtp_decoder_handle`), y diversos mapas que asocian tipos de carga útil con códecs (`payload_map`), gestionan pistas de salida (`outbound_tracks`) y almacenan los últimos fotogramas (`local_frame` y `remote_frame`). Además, mantiene un conjunto de tipos de carga útiles permitidos (`allowed_pts`), asegurando que solo los paquetes RTP válidos sean procesados.
- **`struct VideoFrame` (`video_frame.rs`):** Representa un fotograma de video, incluyendo su ancho, alto, formato (ej. RGB), los bytes de datos de píxeles y una marca de tiempo.
- **`struct H264Encoder` (`h264_encoder.rs`):** Encapsula la codificación de video utilizando la biblioteca `openh264`, configurando parámetros como tasa de fotogramas, tasa de bits y frecuencia de keyframes. Proporciona un método para codificar un `VideoFrame` en un flujo de bytes H.264 y para solicitar un keyframe.
- **`struct H264Decoder` (`h264_decoder.rs`):** Encapsula la funcionalidad de decodificación de video H.264 utilizando `openh264`. Recibe Acces Units (AU) H.264 y las decodifica en `VideoFrame`.
- **`struct H264Depacketizer` (`rtp_session/payload/h264_depacketizer.rs`):** Reconstruye las Acces Units H.264 a partir de los paquetes RTP entrantes
- **`struct H264Packetizer` (`rtp_session/payload/h264_packetizer.rs`):** divide los flujos de bits H.264 en chunks de carga útil RTP para su envío.

## Integración en el proyecto

Su integración en el sistema se basa en un modelo de hilos de trabajo especializados, junto con la gestión coordinada por `core::engine` que invoca periódicamente el método `tick()` para mantener la actividad y el flujo de datos de vídeo en tiempo real.

El funcionamiento interno se articula alrededor de varios **hilos de trabajo dedicados**, cada uno con responsabilidades específicas. El hilo `spawn_camera_worker` se encarga de capturar continuamente fotogramas desde el `CameraManager`, convertirlos desde el formato `OpenCV` a `VideoFrame`s internos y transmitirlos al `MediaAgent` a través del canal `local_frame_rx`. En ausencia de una cámara, este hilo puede generar un patrón de video sintético, garantizando la continuidad del flujo. Por su parte, el hilo `spawn_rtp_decoder_worker` recibe paquetes `RtpIn`, provenientes de `RtpSession`, mediante el canal `rtp_tx`, reconstruye los flujos H.264 con `H264Depacketizer`, y los decodifica en `VideoFrame`s mediante `H264Decoder`. El fotograma resultante se almacena en `remote_frame` para su posterior visualización.

El **manejo de eventos del motor (`EngineEvent`)** es otra parte clave de la integración. `MediaAgent` escucha los eventos provenientes del motor principal de la aplicación para reaccionar ante cambios en el estado de la sesión o en el flujo de medios. Cuando se recibe un evento de establecimiento de sesión (`EngineEvent::Established`), el agente configura las pistas de salida (`outbound_tracks`) con `RtpSession`, habilitando el inicio del envío de video. Asimismo, los eventos `EngineEvent::RtpIn` que contienen paquetes RTP entrantes se reenvían directamente al hilo de decodificación (`rtp_decoder_worker`) para su procesamiento inmediato.

El **ciclo de envío de medios**, implementado en el método `tick(&mut self, session: Option<&Session>)`, es ejecutado periódicamente y constituye el mecanismo central mediante el cual `MediaAgent` mantiene la transmisión activa. Este método comienza llamando a `drain_local_frames()`, que vacía el canal `local_frame_rx` y actualiza el `local_frame` con el fotograma más reciente capturado. Luego, ejecuta `maybe_send_local_frame()`, responsable de preparar y enviar los datos de video al par remoto. Este proceso incluye obtener el último `local_frame`, solicitar un keyframe inicial al codificador H.264 si es necesario, codificar el fotograma con `h264_encoder`, fragmentar los datos codificados en `RtpPayloadChunk`s mediante `h264_packetizer`, y finalmente enviar dichos fragmentos a través de `session.send_rtp_chunks_for_frame()`. Durante este proceso, se actualizan las marcas de tiempo RTP y los contadores de secuencia, asegurando una sincronización correcta del flujo multimedia.

La **interacción con `core::session`** es fundamental para el funcionamiento del módulo. `MediaAgent` obtiene de este componente los `OutboundTrackHandle`s que representan los flujos salientes de medios, los cuales se registran y mantienen actualizados mientras la sesión esté activa. El envío de los datos codificados se realiza invocando los métodos de `core::session`, que delega en `RtpSession` la transmisión real de los paquetes RTP. De manera inversa, los paquetes entrantes son inyectados al `MediaAgent` a través del canal `rtp_tx`, que recibe los datos directamente desde `core::session`.

Finalmente, el módulo expone una interfaz hacia la **interfaz de usuario (UI)** mediante el método `snapshot_frames()`. Este permite obtener los últimos fotogramas locales y remotos almacenados en memoria compartida, posibilitando su visualización en tiempo real. De esta forma, `media_agent` completa el ciclo de captura, codificación, transmisión, recepción, decodificación y renderizado, consolidándose como el componente que materializa el flujo audiovisual de `rustyrtc`.

# Integración entre `RtpSession`, `Session`, `MediaAgent`, `Engine` y `RtcApp`

## Visión general.

La pila se organiza en capas: **RtcApp (GUI)** ↔ **Engine (núcleo y eventos)** ↔ **Session (señalización + transporte)** ↔ **RtpSession (RTP/RTCP)** ↔ **MediaAgent (códecs/frames)**.
El flujo de **control** viaja desde la UI hacia abajo mediante comandos; el flujo de **datos** (medios y estadísticas) sube mediante eventos.

## Responsabilidades por componente.

- **RtcApp (UI/egui)**: muestra estado, logs y video local/remoto; permite pegar/emitir SDP; inicia/termina la llamada. Consume eventos del `Engine` (p.ej., `Established`, `IceNominated`, `Status`, `Error`) y dispara acciones (iniciar oferta, setear respuesta, colgar).

- **Engine**: orquestador central y bus de eventos. Expone una API simple para la UI y encapsula la complejidad del transporte. Mantiene referencias a `Session` y `MediaAgent`. Traduce acciones de la UI en operaciones (crear oferta, aplicar respuesta, cerrar) y enruta **EngineEvents** (p.ej., `Log`, `RtpIn`, `Closing/Closed`).

- **Session**: capa de sesión/peer. Gestiona **SDP** (oferta/respuesta), **ICE** (recolección, nominación y par activo) y el *wiring* del plano de medios. Cuando el par ICE está listo, crea/configura `RtpSession` (bind de sockets, SSRC/PT, timers RTCP). También reenvía/controla señales RTCP de alto nivel (NACK/PLI/BYE) y sincroniza estados con `MediaAgent`.

- **RtpSession**: plano de transporte **RTP/RTCP**. Mantiene `RtpRecvStream`/`RtpSendStream`, procesa UDP entrante (demultiplex RTP vs RTCP), **depacketiza/packetiza**, actualiza métricas (jitter, pérdida, RTT) y emite RTCP compuesto periódicamente (SR/RR/SDES). Entrega *access units* a `MediaAgent` para decodificación y recibe desde `MediaAgent` los *payloads* a enviar.

- **MediaAgent**: frontera de medios. Captura cuadro local (cámara), **encode H.264**, **packetiza** y delega el envío a `RtpSession`. Para entrada remota, **depacketiza** (si aplica), **decode H.264** y publica `VideoFrame` para la UI. Genera/consume señales de control de calidad: solicita **PLI** si necesita *keyframe*; maneja **NACK** (retransmisiones) junto con `RtpSession`. Encapsula mapeo de *payload types* y *tracks*.

## Flujo típico (de extremo a extremo).

1. **Inicio**: RtcApp solicita “conectar”. Engine crea `Session` y prepara `MediaAgent`.

2. **Señalización**: `Session` genera **offer SDP**; RtcApp la comparte; al recibir **answer**, `Session` negocia códecs/PT/SSRC.

3. **Transporte**: ICE nomina un par; `Session` instancia/configura `RtpSession` y señala `Established`.

4. **Salida local**: `MediaAgent` toma frames de cámara → **encode** → `RtpSession/RtpSendStream` → paquetes RTP por UDP.

5. **Entrada remota**: UDP → `RtpSession` clasifica y **depacketiza** → `MediaAgent` **decode** → entrega `VideoFrame` → RtcApp actualiza la vista.

6. **Control**: `RtpSession` envía SR/RR/SDES; procesa RR/PLI/NACK/BYE entrantes. `MediaAgent` pide **PLI** si detecta pérdida de sincronización; `RtpSendStream` atiende **NACK** con retransmisiones.

## Eventos, logs y hilos.

- **Eventos**: todo lo relevante sube al Engine como `EngineEvent` (p.ej., `Status`, `Log`, `RtpIn`, `IceNominated`, `Error`). RtcApp sólo escucha a Engine.

- **Logs**: los componentes aceptan un `LogSink` común; Engine puede reenviar logs a la UI (tap ligero) y a archivo.

- **Concurrencia**: `RtpSession.start()` lanza hilo de recepción UDP y de **RTCP periódico**; `MediaAgent` puede usar un hilo dedicado de **decodificación**. Engine/GUI permanecen reactivos en el hilo de UI.

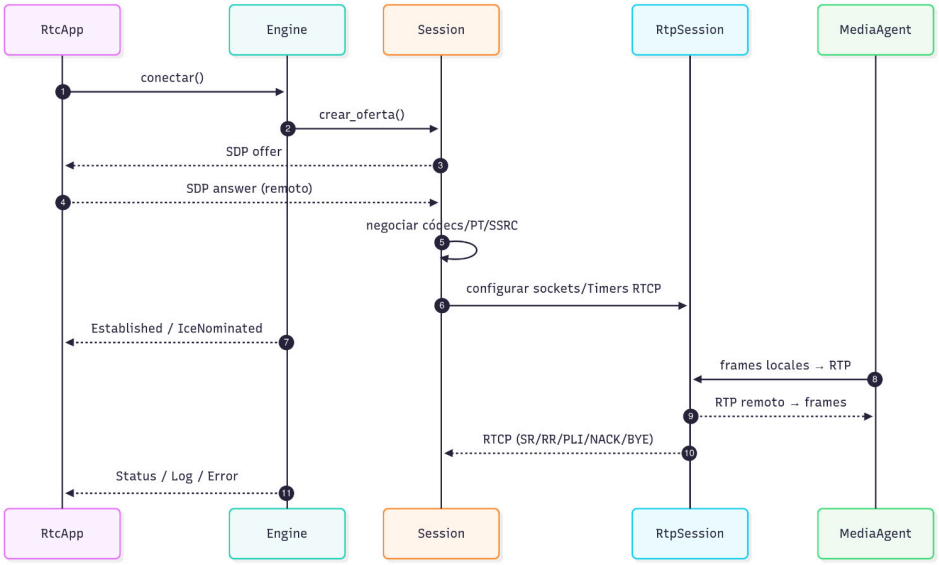
## Cierre y recuperación.

- **BYE** o cierre de socket**:** `RtpSession` desmonta *streams* y notifica `Closing/Closed`.
- **Calidad**: pérdidas elevadas → RR/NACK informan a `RtpSendStream`; **PLI** solicita *keyframe* al remoto.
- **Errores**: cualquier fallo burbujea como `EngineEvent::Error` y la UI decide: reintentar o terminar.

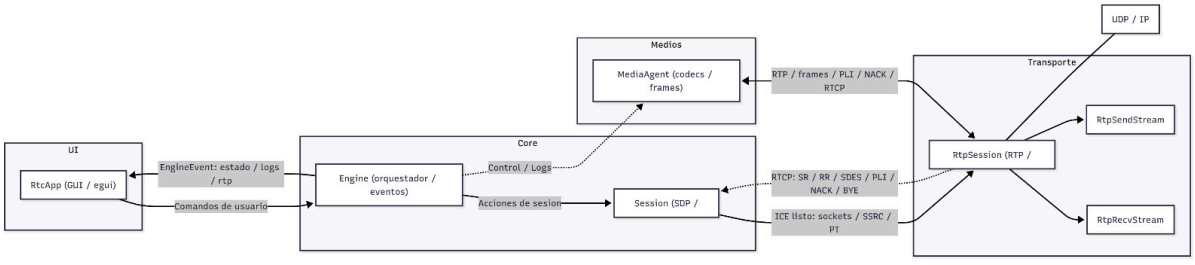
## Beneficios de este diseño.

- **Aislamiento de capas** (UI ↔ orquestación ↔ señalización ↔ transporte ↔ códecs).
- **Backpressure claro** vía RTCP (RR/NACK/PLI) y eventos.
- **Testabilidad**: `RtpSession` y `MediaAgent` se pueden probar en aislamiento; `Engine` se *mockea* desde la UI.
- **Extensible**: añadir audio o más códecs replica el patrón (nuevo *track* en `MediaAgent` + *streams* en `RtpSession`).

## Diagrama de secuencia (flujo extremo a extremo)




## Diagrama de módulos (interacción a alto nivel)



# Conclusiones/aprendizajes de esta entrega intermedia

- Nos fue bastante enriquecedor participar en un proyecto(aunque sea a nivel académico), donde podamos aprender e implementar paso a paso todo lo necesario para lograr una aplicación que simula videoconferencias.
- El hecho de utilizar y aprender nuevas herramientas/librerías que usamos para el desarrollo. Por ejemplo “github projects”, investigar sobre el codec de video, etc.
- La investigación de las diferentes herramientas, nos fue bastante fructífera, a nivel aprendizaje.
- El hecho de trabajar en un proyecto en equipo integrador, separado en spring.
- El feedback semanal y la guía de nuestro corrector, nos oriento bastante.