# Taller de programación

# Entrega final - Room RTC - 2C 2025



| Nombre completo | Padrón | Email |
|---|---|---|
| Gaido Nicolás | 100856 | ngaido@fi.uba.ar |
| Goncalves Rei Tomas | 111405 | TOMAS GONÇALVES REI |
| Olano Soley Nicolás | 104881 | Olano Soley Nicolás Guill… |
| Lamanna Tobias | 104126 | tobias lamanna |

# Índice

# 1. Introducción

## 1.1. Objetivo del proyecto

El objetivo principal del presente Trabajo Práctico es desarrollar una versión en el lenguaje Rust del stack WebRTC, de manera tal que sea posible el desarrollo de una aplicación que permita la realización de videoconferencias entre usuarios en distintos dispositivos. El proyecto debe respetar las specs (i.e., especificaciones), haciendo énfasis en la compatibilidad con clientes existentes de WebRTC. Aspectos no funcionales como la escalabilidad y la eficiencia de la solución deben ser contemplados, ya que el proyecto debe poder ser utilizado por un gran número de personas, desde dispositivos de diversa potencia. El objetivo secundario es desarrollar un proyecto real de software de mediana envergadura aplicando buenas prácticas de desarrollo, incluyendo entregas y revisiones periódicas. La idea es, después de todo, que este proyecto signifique no sólo la aplicación de los temas vistos en la asignatura, sino que, del conjunto de conocimientos aprendidos durante la carrera hasta el momento.

## 1.2. Requerimientos funcionales

Un sistema basado en WebRTC consta de diversos componentes, por lo que será necesaria una planificación y división adecuada del trabajo.
Entre esas componentes podemos encontrar:

- SDP (Session Description Protocol): Utilizado para establecer la conexión entre pares.
- ICE (Interactive Connectivity Establishment): Protocolo de comunicación utilizado entre pares para realizar la transmisión del video.
- Signaling Server: Encargado del discovery de peers. En este caso, el servicio central actuará de signaling server.
- Además, a la hora de transmitir vídeo y sonido, se puede utilizar una variedad de códecs como VP8 y H264 que son los encontrados en la mayoría de implementaciones del protocolo. Pero a nivel de especificación, WebRTC no limita qué códec se debe utilizar, siempre y cuando ambas partes de la conexión se pongan de acuerdo.

## 1.3. Requerimientos no funcionales

Los siguientes son los requerimientos no funcionales para la resolución del proyecto:

- El proyecto deberá ser desarrollado en lenguaje Rust, utilizando las herramientas de la biblioteca estándar.
- Se deben implementar pruebas unitarias y de integración de las funcionalidades que se consideren más importantes.
- El código fuente debe compilar en la versión estable del compilador y no se permite el uso de bloques inseguros (unsafe).
- El código deberá funcionar en ambiente Unix/Linux.
- La compilación no debe generar advertencias del compilador ni del linter Clippy.

- El programa no puede contener ningún busy-wait, ni puede consumir recursos de CPU y/o memoria indiscriminadamente. Se debe hacer un uso adecuado tanto de la memoria como del CPU.
- Las funciones y los tipos de datos (struct) deben estar documentados siguiendo el estándar de cargo doc.
- El código debe formatearse utilizando cargo fmt.
- Las funciones no deben tener una extensión mayor a 30 líneas. Si se requiere una extensión mayor, se debe particionar en varias funciones.
- Cada tipo de dato implementado debe ser colocado en un módulo (archivo fuente) independiente.

### 1.4. Alcance de entrega final

La versión final del sistema integra todos los componentes esenciales de un cliente WebRTC funcional:

- Servidor de señalización real (TCP)
- Autenticacion de usuarios
- Discovery dinámico de peers
- Inicio, aceptación y rechazo de llamadas
- Intercambio automático de SDP Offer/Answer
- Negociación ICE
- Conexión P2P directa
- Transmisión RTP/RTCP
- Cifrado básico mediante DTLS →SRTP
- Interfaz gráfica completa con Lobby y Call

---

## 2. Arquitectura del sistema

### 2.1. Visión general de la arquitectura

**Room RTC** adopta una arquitectura modular en capas donde se separan las responsabilidades de interfaz, codificación multimedia, protocolos de red y transporte. Cada módulo fue diseñado para comunicarse mediante estructuras bien definidas para facilitar tanto la depuración, la extensión futura del sistema y la incorporación de nuevas funcionalidades.
El flujo de datos sigue una dirección descendente, donde la aplicación recibe la entrada del usuario desde la interfaz gráfica, la cual permite conectarse al servidor de señalización. Desde la interfaz gráfica una vez conectado se puede establecer la conexión con un par remoto, lo cual desencadena la inicialización de componentes internos como el módulo de la cámara, los agentes **ICE** y la generación del **SDP** local. Con la conexión establecida las imágenes capturadas por la cámara son codificadas, empaquetadas y finalmente transmitidas hacia el mismo.
El flujo ascendente ocurre en sentido opuesto, los paquetes recibidos desde la red son des-serializados, ensamblados y decodificados, para luego almacenarse. La interfaz gráfica

observa periódicamente este estado y actualiza en tiempo real el panel de video remoto, completando así el ciclo de comunicación bidireccional.

## 2.2. Componentes principales

- Módulo de señalización y conexión (SDP + ICE)
  Este módulo maneja la fase inicial de establecimiento de la conexión. Utiliza el protocolo **SDP** para describir las capacidades multimedia (codecs, puertos, direcciones) e **ICE** para recolectar y probar candidatos de red. **Room RTC** implementa un servidor de señalización completo basado en TCP, encargado de la autenticación, descubrimiento de usuarios, inicio y aceptación/rechazo de llamadas e intercambio automático de SDP Offer/Answer entre los peers. El módulo *signaling_client.rs* maneja el envío/recepción de mensajes estructurados y gestiona el flujo de señalización sin intervención del usuario.

- Módulo de transmisión de audio y video (RTC)
  El módulo **RTC** se encarga de la transmisión en tiempo real mediante **RTP** y el control de calidad con **RTCP**.
  Los frames de video capturados son comprimidos con **H.264**, fragmentados en paquetes **RTP**, enviados por **UDP**, y reconstruidos en el peer remoto. También se implementa un Jitter buffer que permite el guardado temporal de paquetes RTP para su ordenamiento y obtención. Periódicamente, se envían y reciben *Sender Reports (SR)* y *Receiver Reports (RR)*, que calculan jitter, pérdida de paquete y sincronización temporal, y en caso de errores de codificación se envían paquetes PLI para solicitar el envío de un nuevo keyframe que permita recuperar la decodificación de video.

- Módulo multimedia
  - Video: Utilizando el crate **nokhaw** para capturar frames en formato RGB se abstrae el acceso al dispositivo de cámara. Estos frames son comprimidos en formato H.264 utilizando bindings de **OpenH264** y luego paquetizados. El peer remoto recibe los paquetes RTP, los construye y decodifica, generando un *ColorImage* que se renderiza en pantalla.
  - Audio: Utilizamos el crate **cpal** para la captura de audio desde el micrófono. Se utiliza el crate **opus** para el encodeo y desencodeo del audio, que brinda compresión para una transmisión con menor latencia. El par remoto recibe los paquetes RTP, los construye descifra y decodifica, para luego poder reproducirlos con **cpal**.
  - Archivos:

- Módulo de interfaz de usuario
  Implementado con **egui/frame**, permite controlar todas las etapas de la comunicación desde una ventana única, en donde se inicia la cámara, se crea o pega el SDP y se comienza o finaliza una llamada. Esta misma incluye vistas de video local y remoto renderizados como texturas en la superficie gráfica gestionada por egui, respaldada internamente por **OpenGL**, junto con un panel de logs en

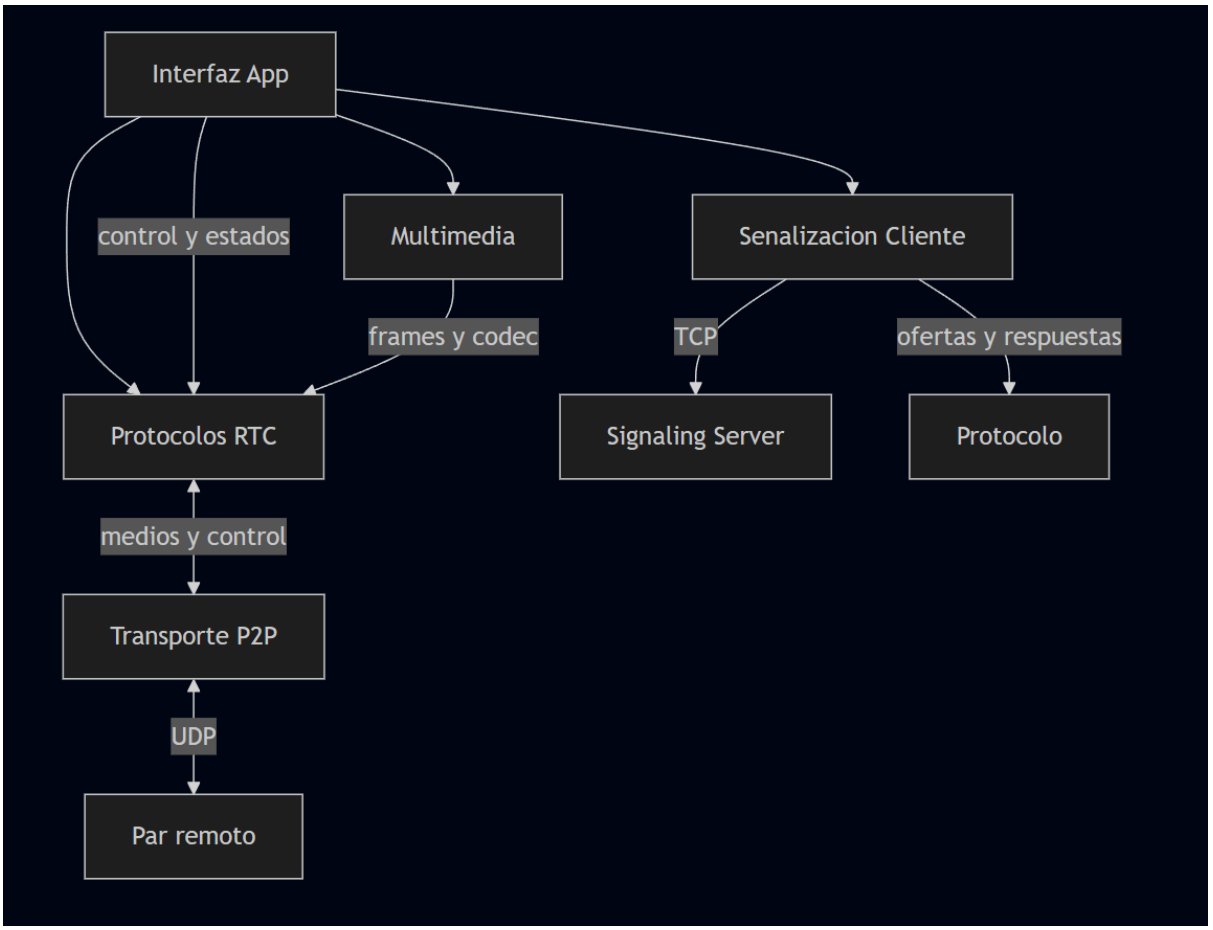
tiempo real con marcas de tiempo.

- Módulo de red y transporte
  Se implementa sobre **UDP** puro de la biblioteca estándar de **Rust**, en donde se gestiona la apertura de puertos locales, envío y recepción de paquetes, y manejo de **STUN Binding Request** para validar conectividad entre peers. Con esto, se permitió un control total sobre el comportamiento de los sockets y tiempos de transmisión.

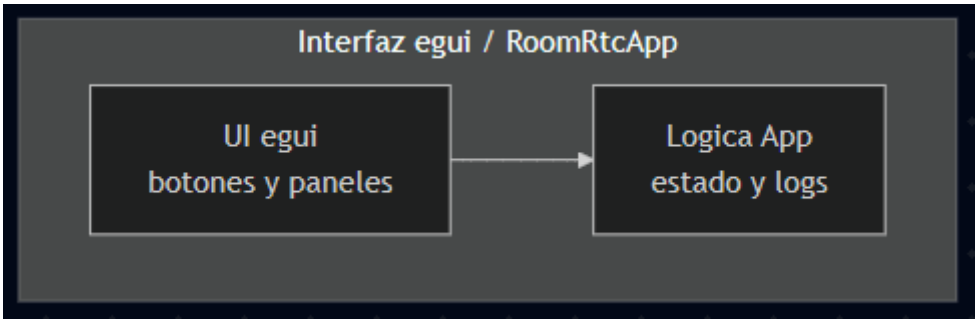
- Módulo de logging.
  Se implementó la persistencia de los logs a archivos mediante un módulo que guarda el nombre del archivo de logs extraído de la configuración. La persistencia se realiza a partir del vector de logs de manera periódica.

- Módulo de autenticación y gestión de usuarios
  Se implementa autenticación real basada en un servidor de señalización. Los usuarios pueden registrarse, iniciar sesión y mantener sesiones persistentes mediante el archivo *users/users.txt*.
  El servidor administra credenciales, sesiones simultáneas y estados, enviando actualizaciones dinámicas a todos los clientes mediante mensajes.
  Además, integra esa información en el flujo de llamadas, actualizando automáticamente el estado de cada usuarios ante diferentes eventos.

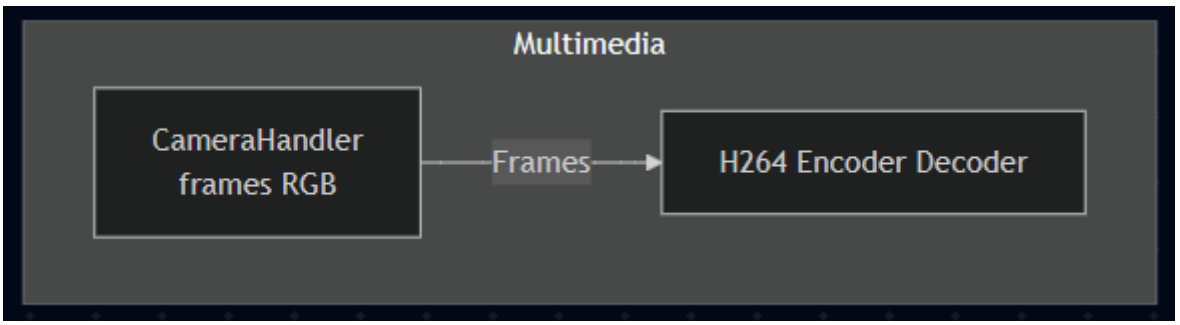
## 2.3. Diagramas de arquitectura



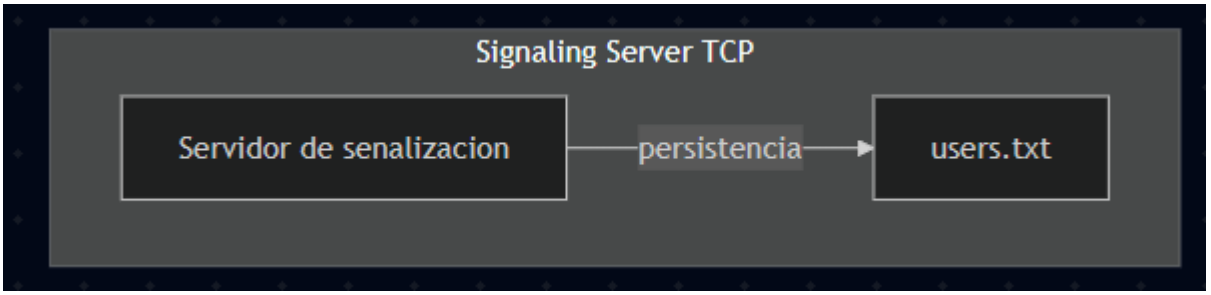

El diagrama muestra las interacciones entre los distintos bloques que componen la arquitectura. A continuación se detallan los módulos pertenecientes a cada bloque:




- UI egui: Renderiza vista de lobby, local y remota; botones y paneles de logs. Actualiza texturas de video y estados sin bloquear el render.
- Lógica de App: Orquesta estados (Lobby/Call), persistencia de logs, manejo de flags de limpieza y conexión. Inicializa ICE, puertos UDP, certificado local y arranque/parada de hilos de envío/recepción. Integra cliente de señalización.

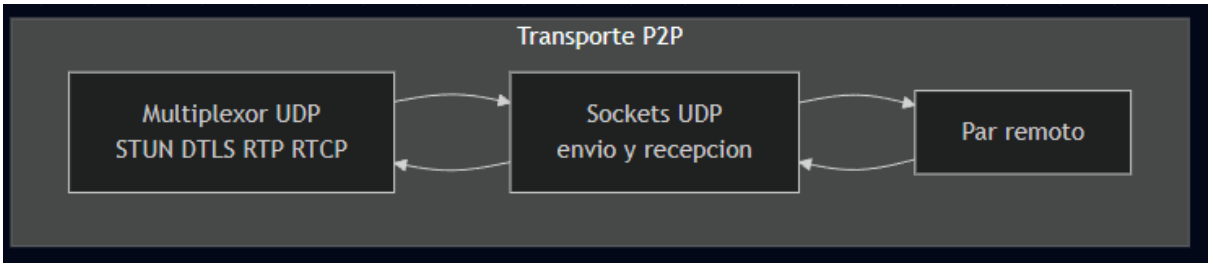



- CameraHandler: Captura frames RGB, expone frames a la App para previsualización y envío.
- H.264 Encoder/Decoder: Convierte RGB a H.264. El encoder permite forzar keyframes ante PLI; el decoder maneja strides y devuelve frames RGB para la UI.

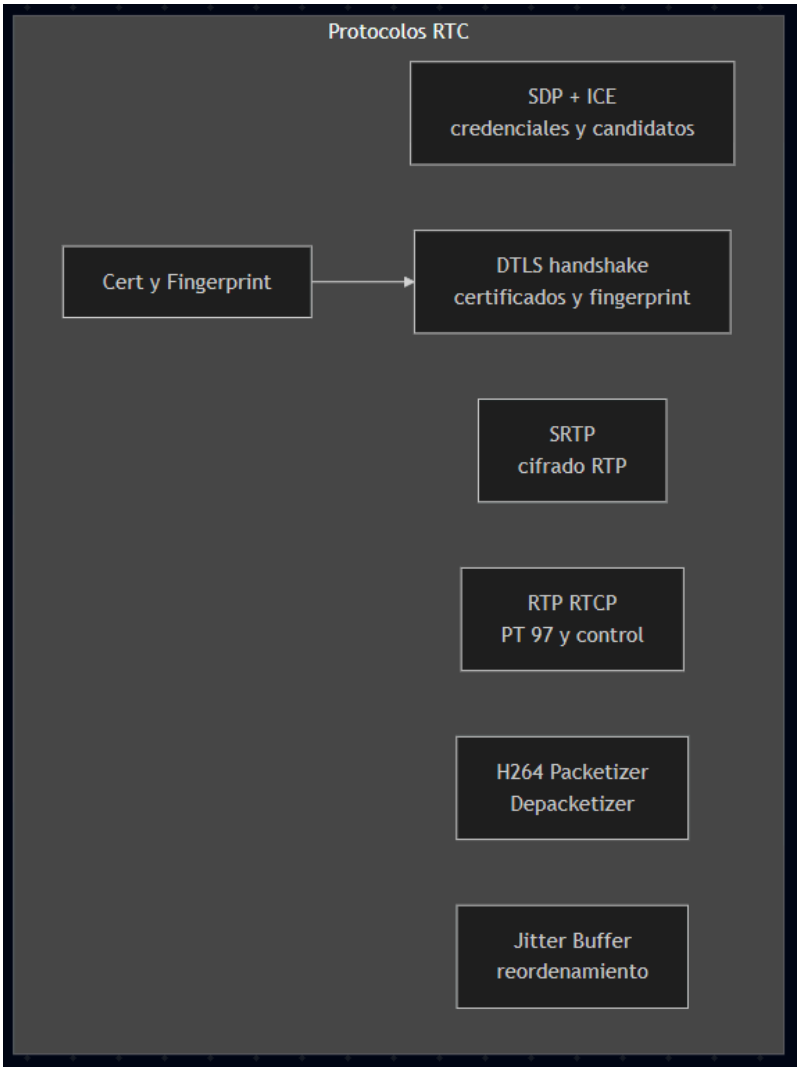



- Signaling Server TCP:
  Administra usuarios, unicidad de sesión, estados (disponible/ocupado/desconectado) y broadcast de la lista. Implementa el protocolo de comando (REGISTER, LOGIN, USER_LIST, INVITE, ACCEPT_CALL, REJECT_CALL, OFFER, ANSWER, SDP, CANDIDATE, END_CALL, LOGOUT).

- Persistencia:
  Archivo plano users/users.txt para registro y carga de credenciales; creación de directorio si no existe.



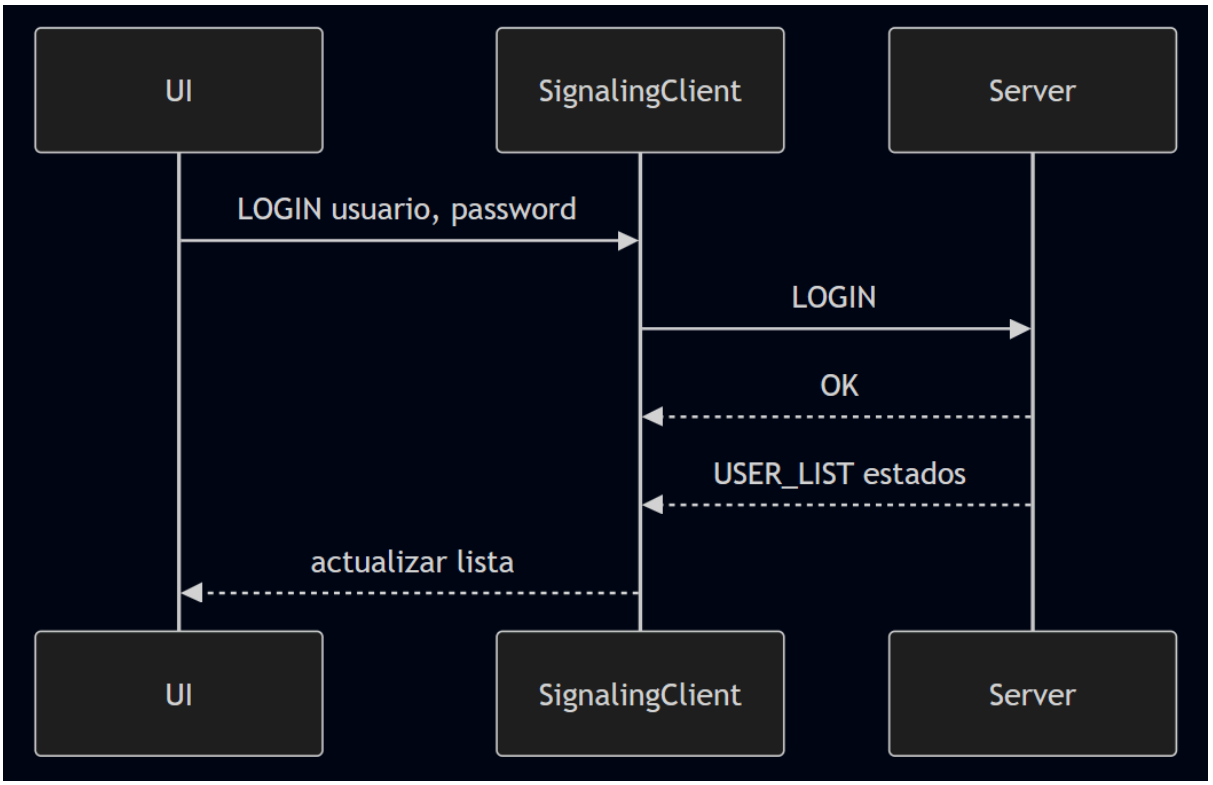

- Multiplexor UDP: Un único UdpSocket clasifica tráfico entrante en STUN/DTLS/RTP/RTCP y lo entrega por canales separados.
- Sockets UDP: Envío/recepción con pacing por paquete, timeouts y control de vida de hilos. Integra SRTP (cifrado de payload) y genera RR/PLI según estado.




- SDP + ICE: Construcción y parseo de SDP con ice-ufrag/pwd, puertos, codecs y fingerprint. ICE recolecta candidatos host y srflx (STUN), hace connectivity checks y selecciona par.
- DTLS: Handshake simplificado con roles según a=setup, validación de fingerprint/certificado y derivación de master_secret (HKDF-SHA256). Retransmisión con backoff.
- SRTP: Cifrado/desencriptado de payload RTP
- RTP/RTCP: Header RTP y paquetización H.264 (FU-A), reensamble y entrega por Access Units. RTCP SR/RR/SDES/BYE y PLI para solicitar keyframes. Estado de recepción y métricas.
- Packetizer/Depacketizer: Divide y une NALUs (SPS/PPS/IDR, FU-A), marca fin de frame con bit Marker y construye AU para el decoder.
- Jitter Buffer: Reordenamiento por sequence number. Guarda paquetes RTP en orden.
- Cert y Fingerprint: Genera X.509 auto-firmado (rcgen) y calcula/verifica fingerprint SHA-256 para SDP/DTLS.

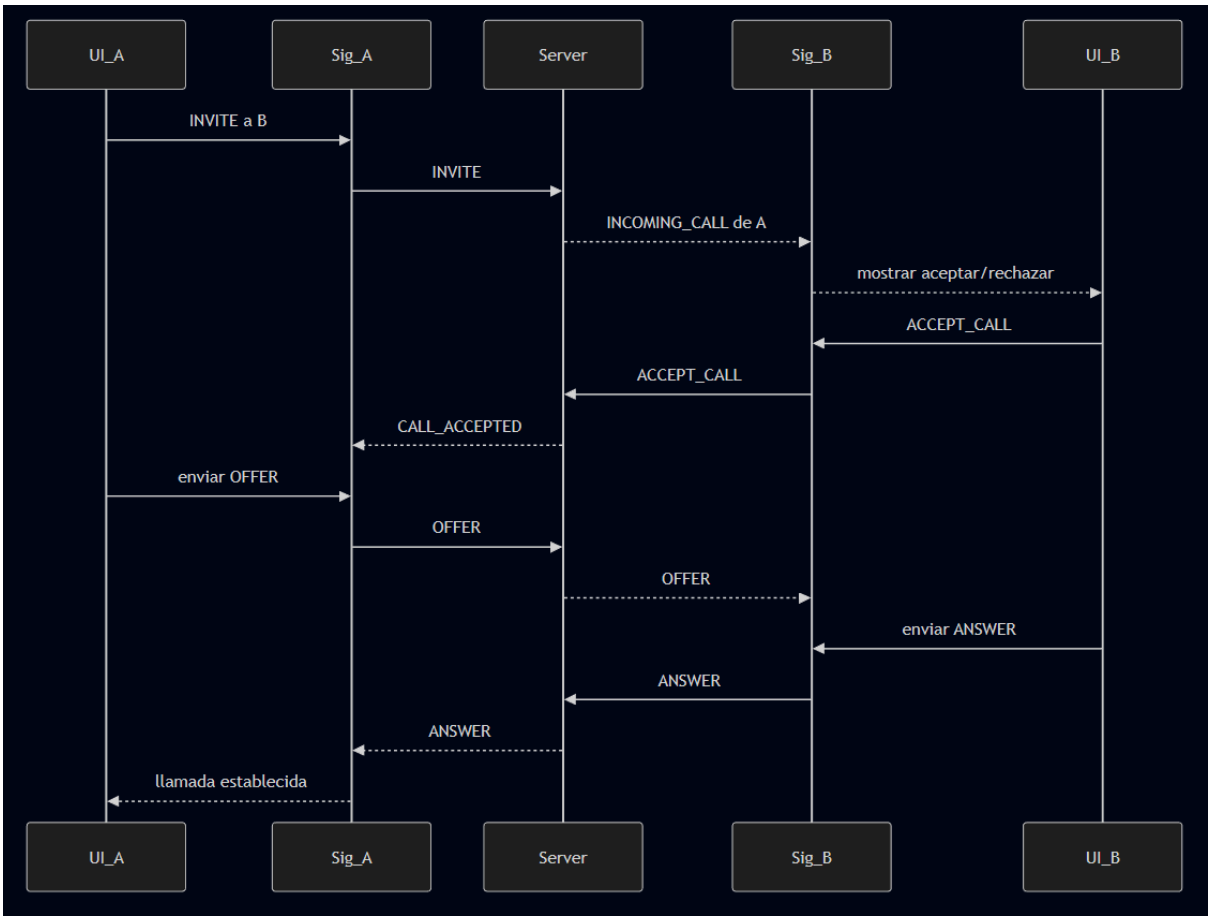
## 2.4. Diagramas de secuencia

### 2.4.1. Autenticación y lista



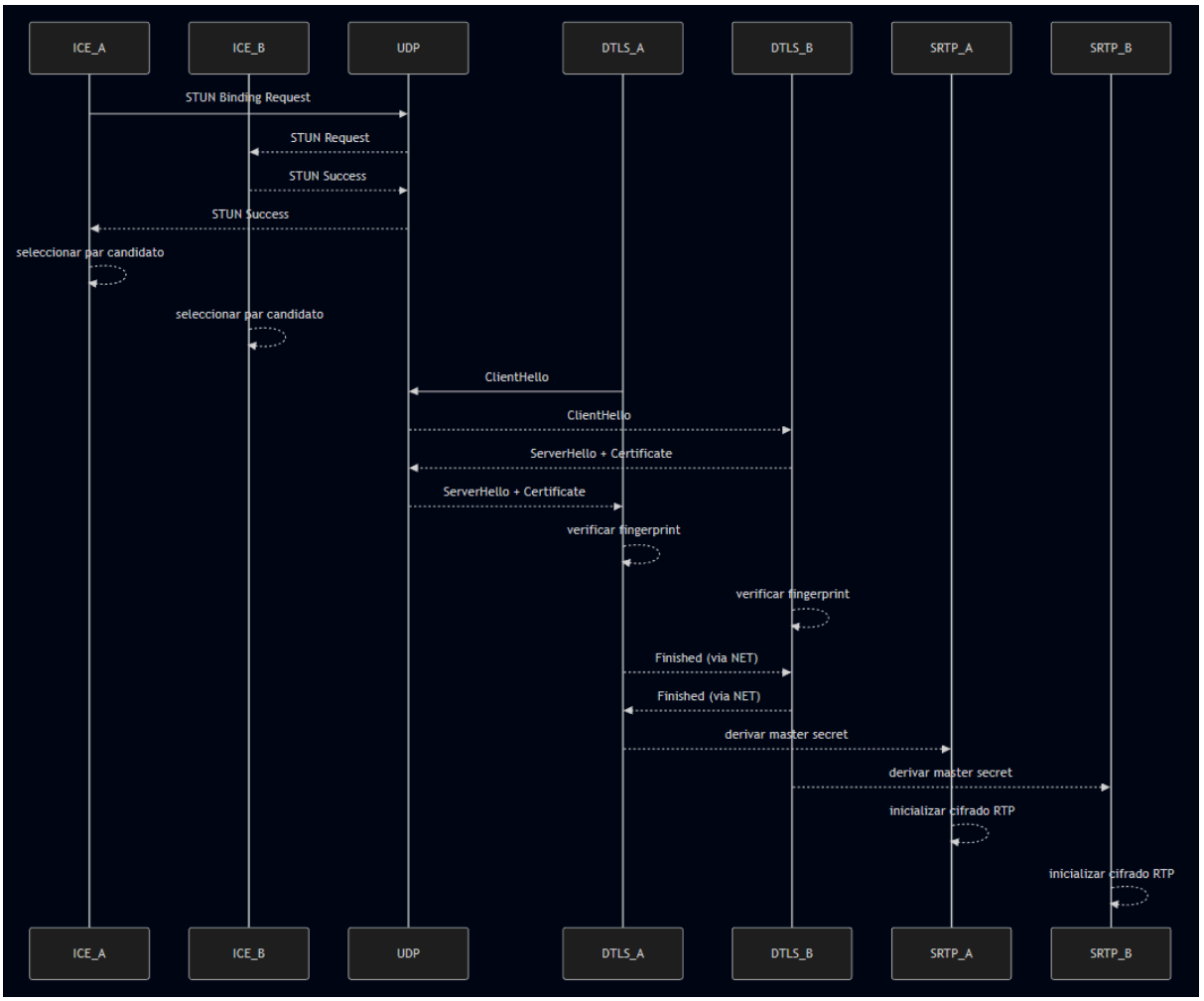

El usuario ingresa credenciales en la UI y el cliente de señalización envía un LOGIN al servidor usando el protocolo con framing; el servidor valida, responde OK y emite un USER_LIST con los usuarios y estados actuales; el cliente actualiza la UI con esa información, dejando al usuario listo para iniciar o recibir llamadas.

### 2.4.2. Llamada y Offer/Answer



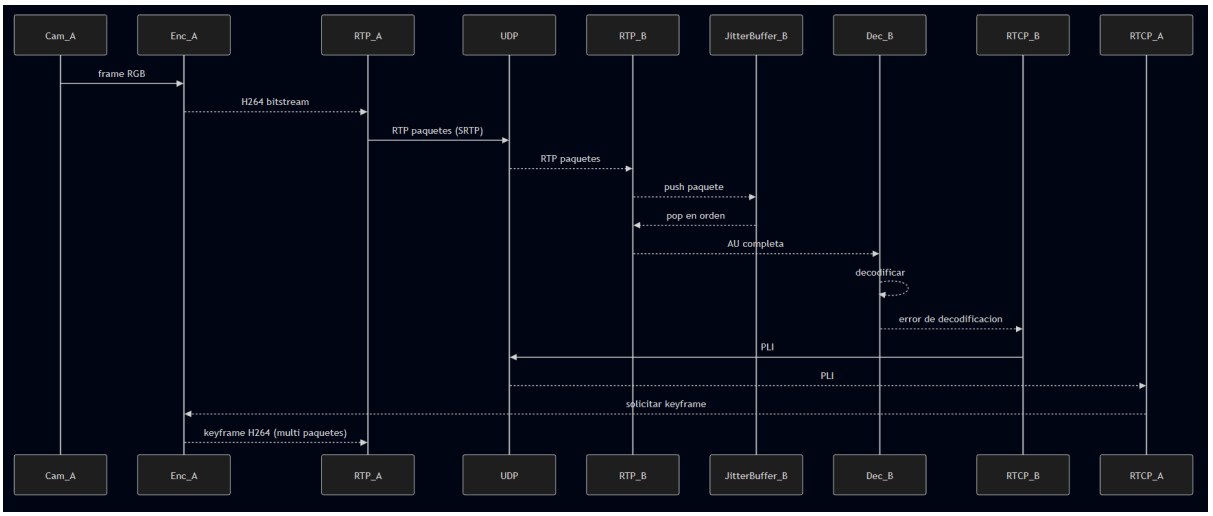

El cliente A envía un INVITE al servidor, que notifica a B con INCOMING_CALL; B acepta (ACCEPT_CALL), el servidor confirma a A y comienza el intercambio de descripciones de sesión: A envía la OFFER a través del servidor, B la recibe y responde con una ANSWER por el mismo canal; al recibir la ANSWER, A completa la negociación, la UI marca la llamada como establecida y ambos quedan listos para pasar a conectividad P2P.

### 2.4.3. Conexión ICE + DTLS + SRTP



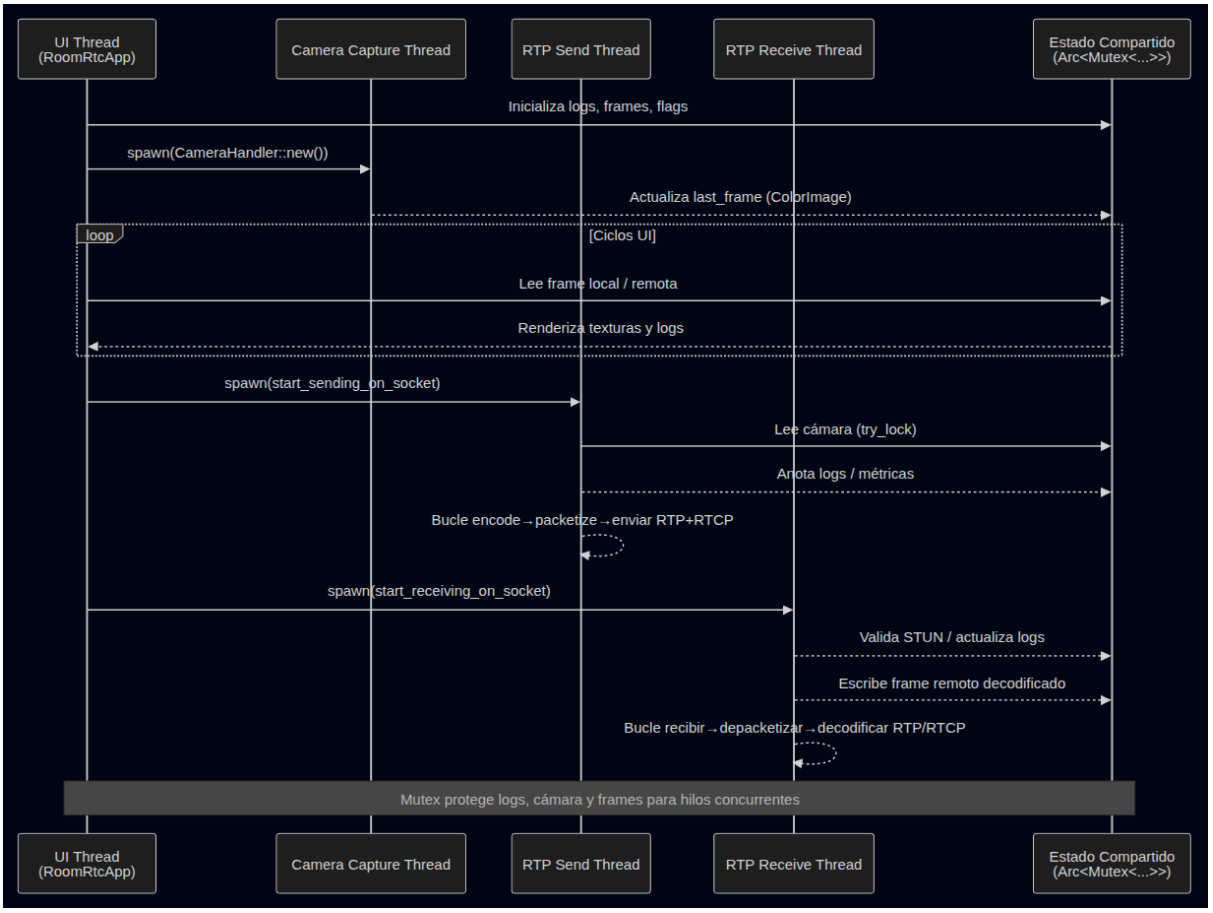

Tras intercambiar SDP, los agentes ICE de ambos lados ejecutan STUN (Binding Request/Success) para verificar conectividad y seleccionar el par de candidatos; con la ruta UDP establecida, inicia el handshake DTLS (ClientHello/ServerHello/Certificate/Finished), ambos validan fingerprint/certificados, derivan el master secret y configuran SRTP; con esto queda lista la encriptación punto a punto del tráfico RTP/RTCP.

### 2.4.4. Transmisión Media + PLI




El emisor captura un frame, lo codifica en H.264 y lo envía como paquetes RTP (cifrados con SRTP) por UDP; el receptor reordena con el jitter buffer, reensambla Access Units y las decodifica para renderizar; si hay pérdida o corrupción que impide decodificar, el receptor envía un RTCP PLI solicitando un keyframe; al recibirlo, el emisor fuerza un IDR, lo que restaura la decodificación y recupera la continuidad del video recibido.

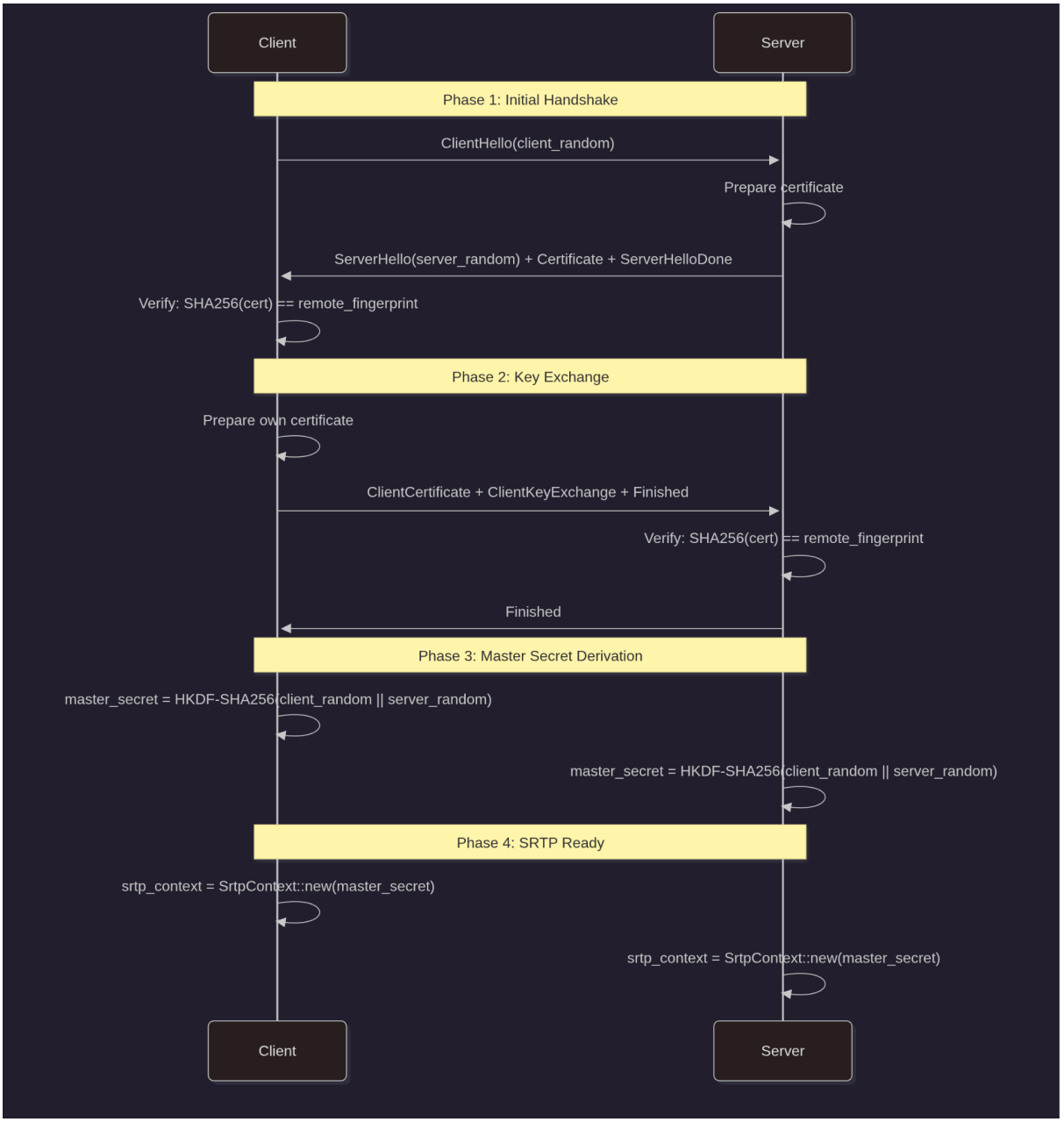
### 2.4.5. Modelo de concurrencia




El esquema ilustra la interacción entre los distintos hilos del sistema: la interfaz principal (UI Thread), la captura de cámara, y los hilos de envío y recepción **RTP**.

La aplicación lanza (*spawn*) los hilos encargados de captura y transmisión (*start_sending_on_socket*) y de recepción (*start_receiving_on_socket*), mientras la UI renderiza los frames locales y remotos de forma continua.
El acceso concurrente a recursos compartidos como logs, cámara y frames se controla mediante Arc<Mutex<...>>, garantizando seguridad en memoria y consistencia entre los hilos.

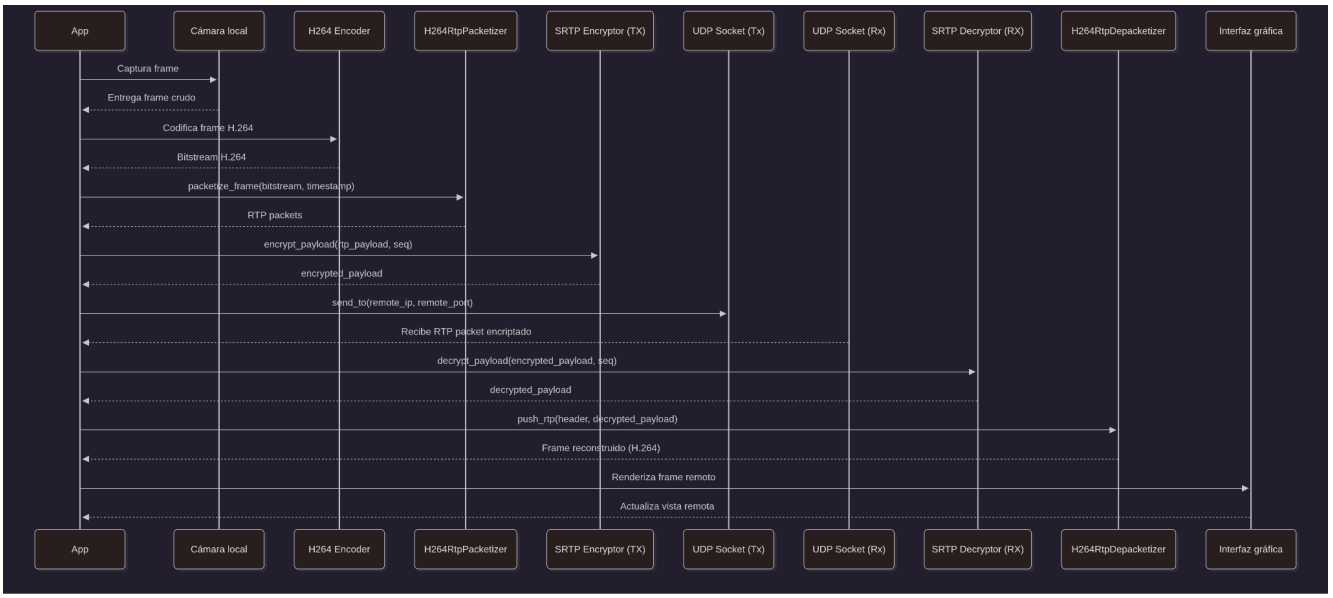
### 2.4.6 Handshake DTSL




El diagrama ilustra el proceso completo de establecimiento de una comunicación segura mediante DTLS (Datagram Transport Layer Security) y la posterior inicialización del contexto SRTP utilizado para proteger el tráfico multimedia.
En las fases 1 a 2 vemos el intercambio de mensaje que se da entre cliente y servidor para compartir sus certificados y la información adicional (client random + server random) que se utilizará luego para inferir el master secret.

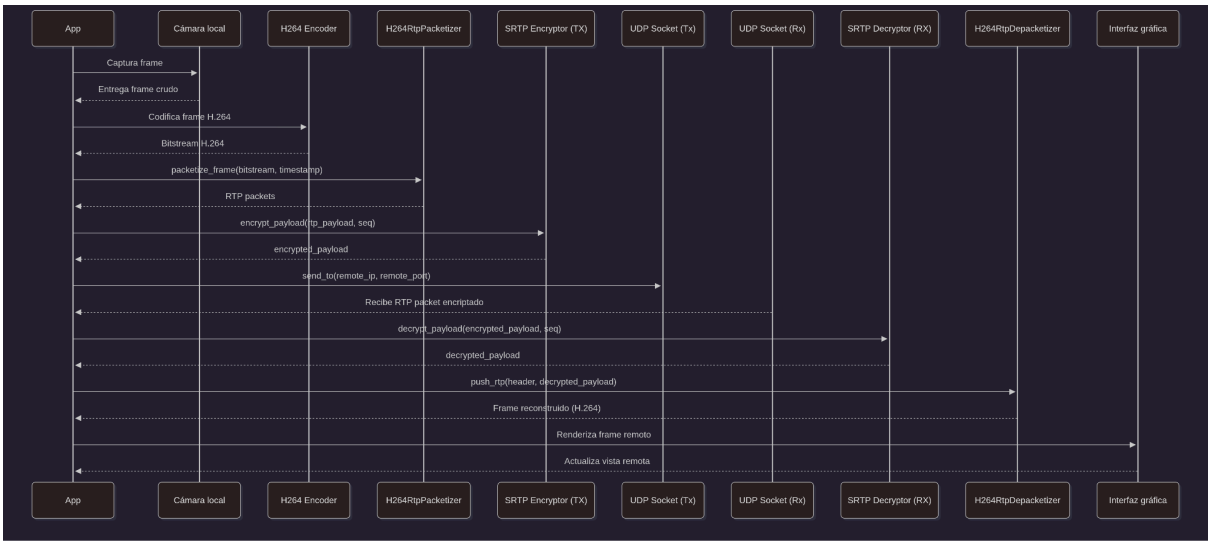
Una vez finaliza el handshake, ambos pares verifican que los certificados son correctos (su hash coincide con el fingerprint compartido en el SDP). Luego se deriva el master secret utilizando los parámetros de clave negociados en el handshake.
Finalmente se inicializa el contexto SRTP que permitirá encriptar y desencriptar los paquetes RTP.

### 2.4.7 Transmisión de video con encriptado




El diagrama muestra el proceso de envío y recepción de video en tiempo real: la cámara local captura frames **RGB** que son codificados en **H.264**, empaquetados en tramas **RTP**, luego encriptados y finalmente enviados por **UDP**.
En el extremo remoto, los paquetes **RTP** se re ensamblan, desencriptan y decodifican, reconstruyendo el video que se actualiza en la interfaz gráfica aproximadamente cada 33 ms (*~30 fps*).

### 2.4.8 Transmisión de audio con encriptado




El diagrama muestra el proceso de envío y recepción de audio en tiempo real: el micrófono captura audio que se codifica en **OPUS**, empaquetados en tramas **RTP**, luego encriptados y finalmente enviados por **UDP**.
En el extremo remoto, los paquetes **RTP** se re ensamblan, desencriptan y decodifican, reconstruyendo el audio que se sincroniza con el video y reproduce con **cpal.**

## 2.5. Flujo de comunicación entre peer y peer

1. **Inicialización local**:
   El peer detecta sus candidatos locales mediante el crate if-addrs, y los candidatos STUN mediante STUN request a un servidor público. Asigna puertos libres (audio y video), y se guardan los candidatos ICE y credenciales (*ufrag*, *pwd*).
2. **Señalización previa a la conexión P2P:**
   Antes de comenzar el proceso de ICE/STUN, los peers deben descubrirse y negociar roles (caller/callee). Los SDP offer y answer se envían por medio del signaling server, y en caso de resultar positivo el intercambio se procede al envío de los mensajes STUN para establecer la conexión entre pares.
3. **Oferta y respuesta SDP:**
   El primer peer crea una *Offer SDP* que incluye sus parámetros ICE y RTP. El segundo peer genera la *Answer SDP* y la copia de regreso al primero por medio del signaling server.
4. **Comprobación ICE (STUN):**
   Ambos peers intercambian *Binding Request* por cada par de candidatos para verificar la conectividad directa. Si la prueba es exitosa, se establecen los sockets UDP finales para RTP/RTCP. El cliente que crea la offer toma el rol controlling y realiza los chequeos activos, y el cliente que genera la answer toma el rol controlado que implementa respuesta para STUN binding request pero no genera consultas propias.
5. **Handshake DTSL**:
   A través del handshake, se valida la identidad de cada uno de los pares, verificando

si el hash del certificado intercambiado coincide con el compartido en el SDP en las instancias previas. A su vez, en el handshake se intercambian valores randomizados que permiten inferir un master secret que se utilizara luego para cifrar y descifrar los paquetes enviados.

6. **Transmisión multimedia:**
   Se capturan frames locales, se codifican, se fragmentan en RTP, se cifran y se envían al peer remoto. En paralelo, el receptor depaquetiza, descifra, decodifica y renderiza el audio + video recibido.
7. **Control y medición:**
   Se intercambian paquetes RTCP (*SR/RR*) que informan estadísticas de rendimiento y pérdida.

## 2.6. Patrones de diseño utilizados

Room RTC emplea una combinación de patrones de diseño estructurales y de concurrencia que favorecen la separación de responsabilidades, la extensibilidad del sistema y la seguridad en el acceso a recursos compartidos.

La arquitectura sigue un esquema similar al MVC(Modelo-Vista-Controlador), en donde la vista es gestionada por la interfaz egui, que renderiza los videos y los paneles interactivos; el controlador reside en el archivo app.rs, el cual recibe los eventos de la UI, ejecuta las acciones correspondientes y actualiza los estados compartidos; y el modelo está compuesto por los modulos logicos que encapsulan los datos y comportamientos de cada dominio.

Con este enfoque se mantiene separada la lógica del rendering, simplificando las pruebas y la depuración.

Los hilos dedicados a envío y recepción de video funcionan como un patrón de **Productor/Consumidor**, en donde la cámara produce frames continuamente y los hilos de red los consumen, procesan y transmiten.

El intercambio de información se realiza mediante estructuras sincronizadas (*Arc<Mutex<T>>*), lo que garantiza seguridad sin bloqueos prolongados en la UI.

Cada protocolo o componente está implementado como un módulo independiente, lo cual facilita la reutilización y la sustitución de componentes sin modificar las capas superiores. Se aplica un patrón de *resiliencia* mediante el manejo centralizado de errores para que, en lugar de detener el programa ante fallos, cada error se encapsula y se reporta visualmente en el *panel de logs*. De esta manera, el sistema mantiene su funcionamiento aunque una operación de red, cámara o codec falle temporalmente.

## 2.7. Subsistema de señalización (Cliente)

La señalización es el mecanismo que permite que los usuarios se autentiquen en el sistema, descubran a otros peers disponibles y coordinen el inicio de una llamada WebRTC. Room RTC implementa un protocolo de señalización sobre TCP, compuesto por un **servidor central** y un **cliente de señalización** integrado en la aplicación principal.

El módulo *signaling_client.rs* encapsula toda la logica de comunicación con dicho servidor, proporcionando una API de alto nivel para enviar y recibir mensajes estructurados a través de un canal confiable.

### 2.7.1. Arquitectura interna del cliente

Al establecer una conexión con el servidor, el cliente crea:

- Un **thread de envío**, responsable de serializar mensajes mediante *build_message()* y enviarlos en el formato enmarcado con *write_framed()*.
- Un **thread de recepción**, que lee mensajes del socket mediante *read_framed()*, los parsea con *parsed_message()* y los entrega a la aplicación a través de un canal *mpsc*.

Ambos hilos comparten memoria únicamente mediante canales seguros y locks, garantizando concurrencia libre de data races.

### 2.7.2. Protocolo de mensajes

El cliente implementa métodos específicos para cada tipo de mensaje del protocolo:

- Autenticación: *login()*, *register()*, *logout()*.
- Administración de usuarios: *list_users()*.
- Llamadas:
    - *invite()*, *accept_call()*, *reject_call()*
    - *send_offer()*, *send_answer()* para intercambio de SDP
    - *end_call()* para terminar sesiones activas

Cada mensaje se construye de manera determinística, el servidor procesa estos mensajes y distribuye eventos a los distintos clientes mediante el mismo formato.

### 2.7.3. Integración con la interfaz gráfica

El cliente de señalización no solo envía y recibe mensajes, sino que se integra directamente con la lógica de la interfaz. El ciclo completo de señalización desde la UI es el siguiente:

1. El usuario se registra o inicia sesión (*LOGIN/REGISTER)*
2. La UI recibe mensajes *OK* o *ERROR* mediante *try_recv()*
3. Se actualiza la lista de usuarios con *USER_LIST*
4. El usuario envía *INVITE*
5. La UI del peer remoto recibe *INCOMING_CALL*
6. El peer acepta (*ACCEPT_CALL*) o rechazar (*REJECT_CALL*)
7. Si la llamada se establece, se inicia el intercambio de SDP (*OFFER → ANSWER*)
8. El UI cambia de pantalla *Lobby → CallApp* de forma automática

El diseño evita bloqueos del thread principal, ya que toda comunicación con el servidor ocurre en hilos dedicados gestionados por *Signalingclient.*

## 2.8. Subsistema de señalización (Servidor)

La arquitectura de Room RTC incorpora un servidor central de señalización que actúa como punto de encuentro entre los usuarios, gestionando la autenticación, las listas de usuarios, las solicitudes de llamada y el intercambio inicial de SDP entre los peers.

### 2.8.1. Responsabilidades del servidor

1. Autenticación de usuarios:
   1.1. Registro persistente de usuarios
   1.2. Validación de credenciales en *LOGIN*
   1.3. Rechazo de usuarios duplicados o conectados simultáneamente
2. Gestión del estado global del sistema:
   2.1. Tabla de usuarios registrados
   2.2. Tabla de usuarios online con su Tcp Stream asociado
   2.3. Tabla de estados: *disponible, ocupado, desconectado*
3. Difusión de la lista de usuarios:
   3.1. Cada cambio en el estado de un usuario provoca un broadcast con el formato:
   *USER_LIST|list=user1:estado;user2:estado;...*
4. Administración de llamadas*:*
   4.1. Entre usuarios, invitación, aceptación, rechazo o finalización de llamadas; con transición automática de estados ( *ocupado ←→ disponible)*
5. Reenvío de mensajes de WebRTC:
   5.1. Reenvío de *OFFER/ANSWER/SDP/CANDIDATE*
   5.2. El servidor no interpreta el contenido de los mensajes SD, solo se limita a reenviarlos al destinatario correspondiente para completar el flujo *ICE/SDP*

El servidor utiliza un modelo de **thread por conexión**, donde cada cliente es atendido por su propio hilo. Para compartir el estado global se utiliza *Arc<Mutex<ServerState>>,* lo que permite accesos concurrentes serializados, manipulación segura de la lista de usuarios y emisión de broadcast coherentes.

Los usuarios registrados se almacenan en *users/users.txt* donde cada línea contiene *username,password*. Esta información se carga al inicio mediante *load_users()* y se actualiza con *save_user()*.

Este servidor actualiza automáticamente el estado de ambos usuarios al recibir:

| **Evento** | **Estado *from*** | **Estado *to*** |
|---|---|---|
| INVITE | ocupado | ocupado |
| ACCEPT_CALL | ocupado | ocupado |
| REJECT_CALL | disponible | disponible |
| END_CALL | disponible | disponible |

Permitiendo que otros clientes visualicen en tiempo real qué usuarios pueden recibir llamadas.

## 2.9. Interfaz Gráfica y Flujo de Interacción

La aplicación Room RTC incluye una interfaz gráfica desarrollada con **egui/eframe**, que sirve como capa de control para toda la lógica de señalización y comunicación multimedia. La UI se encuentra dividida en dos pantallas principales: **Lobby** y **Call**, administradas por **RoomApp.**

Componentes principales:

- **Lobby App**: Vista inicial utilizada para autenticación en el servidor, visualización de usuarios conectados e inicio o recepción de llamadas.
- **CallApp**: Pantalla de llamada activa donde se muestra vista local y remota, se gestiona SDP/ICE según el rol (caller/callee), se controla el envío de la SDP inicial y se permite colgar la llamada.
- **UI de videollamada**: Renderización de texturas, logs, RTCP stats y botones de control.

### 2.9.1. Flujo del Lobby

Se implementa una máquina de estados que controla la sesión del usuarios y coordina la señalización con el servidor. El flujo es el siguiente:

- **Autenticación:**
  - El usuario completa usuario/contraseña
  - El cliente crea un *SingnalingClient*
  - Envía *LOGIN* o *REGISTER*
  - El servidor responde con *OK* o *ERROR*
  - Lobby App actualiza su estado en base a estos mensajes
- **Actualización de usuarios conectados:**
  Cada vez que llega *USER_LIST,* Lobby App:
  - Parsea la lista
  - Actualiza *Arc<Mutex<Vec<(String, String)>>>*
  - Actualiza la UI en tiempo real
  - Detecta si un usuario se desconecte durante la llamada, volviendo a la pantalla de Lobby
- **Inicio de llamada (*caller*):**
  Cuando el usuario clickea *Llamar*:
  - Se envía *INVITE* al servidor
  - Lobby App pasa al estado *is_caller = true*
  - Se muestra "*Llamando a X…*"
- **Recepción de llamada (*callee*):**
  Al recibir *INCOMING_CALL*:
  - La UI muestra los botones de *"Aceptar" / "Rechazar"*
  - Si acepta envía *ACCEPT_CALL* y el Lobby App cambia a pantalla de llamada
  - Si rechaza envía *REJECT_CALL* y el estado vuelve a disponible

### 2.9.2. Flujo de pantalla de llamada

Se administra la lógica de llamada directa entre los dos peers, el flujo es el siguiente:

- **Inicialización**:
  - Al entrar al CallApp, se inicia la camara y si es caller se crea la SDP Offer y se envía automáticamente al peer remoto vía signaling
- **Recepción de SDP Offer/Answer:**
  - Se procesa el mensaje, si es *OFFER* el calle genera el *ANSWER* automáticamente y si es *ANSWER* el caller completa la negociación.
- **Finalización de llamada:**
  - Recibe *CALL_ENDED y* vuelve al Lobby; si envia *END_CALL* al servidor, vuelve al Lobby también.

### 2.9.3. Render de video y panel de control

El archivo *ui.rs* implementa el render local, el render remoto, el botón de “Colgar”, logs, logs RTCP y actualización continua. Todo el pipeline está desacoplado de la UI, como el thread de captura de cámara y el thread de recepción/envío RTP, la UI solo renderiza texturas. Adicionalmente, los logs se persisten en un archivo para facilitar el debugging.

## 2.10. Protocolo de Mensajes

Para la comunicación entre el cliente de señalización y el servidor central se diseñó un protocolo textual, implementado en el módulo *message.rs*. Este protocolo es independiente de WebRTC y se utiliza exclusivamente para operaciones de autenticación, descubrimiento de usuarios e intercambio de SDP.

Cada mensaje enviado por TCP posee la siguiente estructura:
*<length>\n*
*TYPE|key=value|key=value|...*

Esta estrategia de framing asegura que los mensajes pueden contener cualquier carácter sin ambigüedad (excepto **|** y **=** como separadores), no existe dependencia de delimitadores arbitrarios y se evitan lecturas incompletas o concatenación accidental de mensajes.

El envío se realiza mediante *write_framed()*, que añade un prefijo con la longitud exacta del mensaje ( *<length>\n<message>* ) permitiendo que el receptor conozca **exactamente** cuántos bytes debe leer antes de procesar el mensaje.
Del lado del receptor, la función *read_framed()* lee la línea de tamaño, para el entero y luego lee exactamente esa cantidad de bytes del stream TCP. De esta forma evitamos fragmentación de mensajes, concatenación del mismo paquete, lecturas parciales o bloqueos indefinidos.
El texto leído se convierte en un *Message* a través del parseo, en donde se separa en *msg_type* que identifica el comando y *fields* que es la tabla de clave → valor. Con esto se permite al servidor y al cliente manipular los mensajes como estructuras claras y tipadas.

## 2.11. Multiplexado de tráfico UDP

Room RTC implementa un módulo de multiplexado de paquetes UDP inspirado en el comportamiento real de WebRTC, donde todos los protocolos viajan por un único socket. Esto permite simplificar la administración de puertos, reducir el overhead de sincronización y centralizar la lógica de recepción.
El modulo *multiplexer.rs* ejecuta un hilo dedicado que escucha en un único Udp Socket y clasifica cada paquete según sus primeros bytes en **STUN**, **DTLS**, **RTP, RTCP** y **SCTP,** enruta cada paquete a un canal independiente y los hilos de ICE, DTLS, RTP y DataChannel consumen únicamente su tipo de paquete.

## 2.12. Subsistema de transferencia de archivos

Room RTC implementa un sistema completo de transferencia de archivos sobre **WebRTC Data Channels**, utilizando el protocolo **SCTP** para transmisión confiable de datos durante una llamada activa.

### 2.12.1. Arquitectura de Data Channels

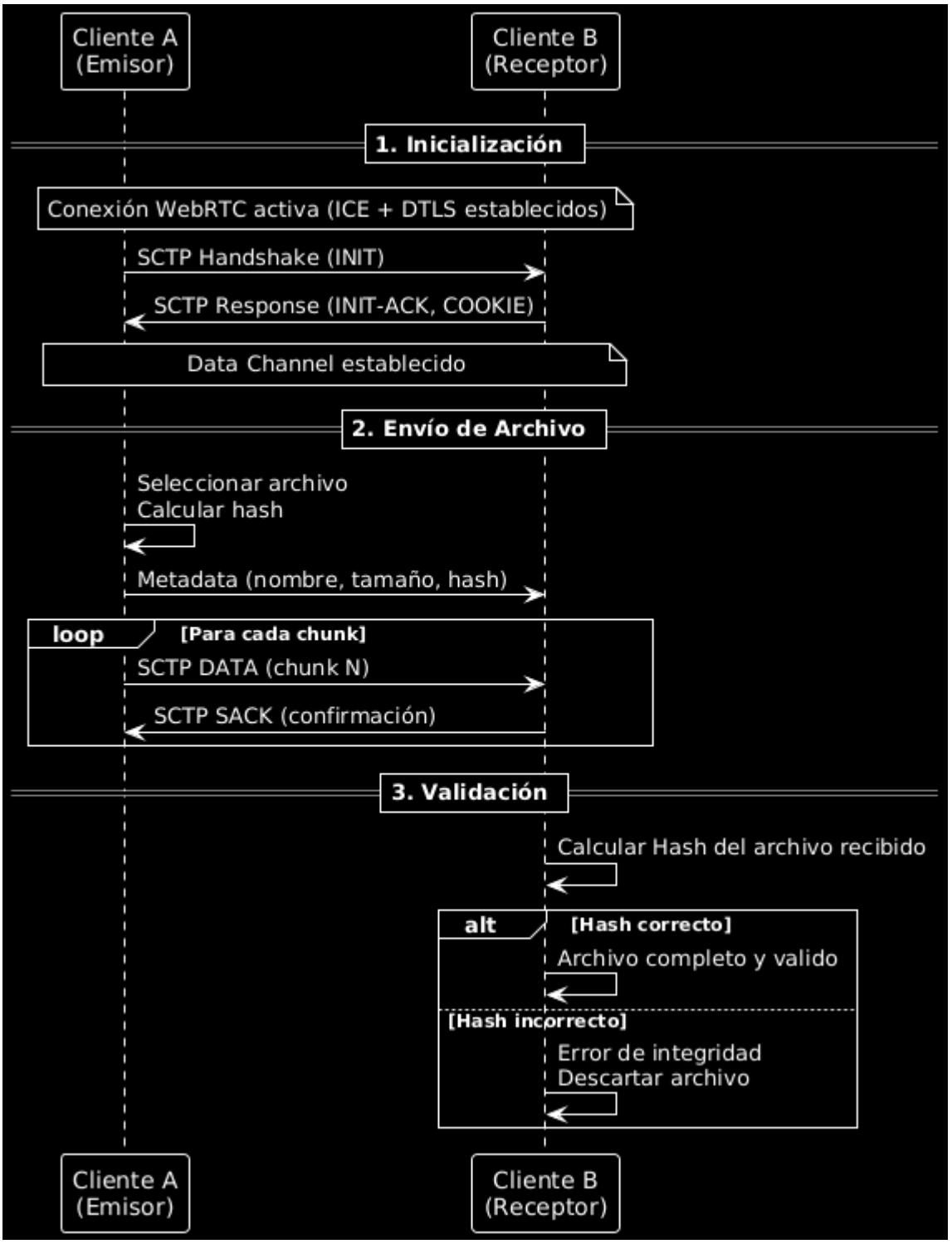
**DataChannelManager** (*data_channel.rs*)
Componente central que encapsula la lógica de **SCTP** utilizando el crate *sctp_proto* y proporciona una API para crear y gestionar canales de datos.

**Responsabilidades:**

- Mantiene el **Endpoint** SCTP y la **Association** activa con el peer remoto
- Gestiona el ciclo de vida de múltiples data channels simultáneos
- Administra stream IDs (pares para *initiator*, impares para *acceptor*)
- Procesa eventos SCTP: Connected, Stream, AssociationLost
- Encola transmisiones pendientes para envío por UDP
- Recolecta datos recibidos organizados por stream_id

**Estados de un Data Channel:**
- **Connecting**: Canal iniciando handshake SCTP
- **Open**: Canal establecido y listo para transmitir
- **Closed**: Canal cerrado o asociación perdida

2.12.2. Flujos de envio y recepción



**Flujo de envío:**

1. Usuario selecciona archivo desde UI
2. *send_file(path)* lee el archivo completo del filesystem
3. Valida límite de tamaño configurado
4. Calcula SHA-256 del contenido
5. Crea *FileMetadata* (nombre, tamaño, hash)
6. Crea data channel dedicado con *create_channel()*
7. Serializa y envía metadata como primer mensaje
8. Divide archivo en chunks según *chunk_size_kb*
9. Envía cada chunk secuencialmente con *send_file_data()*
10. Registra estado en *active_uploads*

**Flujo de recepción:**

1. *poll_incoming()* consulta datos de todos los streams
2. Para cada stream con datos:
    a. Primer mensaje: deserializa *FileMetadata*
    b. Crea *DownloadState* con buffer de acumulación
    c. Mensajes subsiguientes: acumula chunks
3. Al recibir todos los bytes esperados:
    a. Calcula SHA-256 del contenido recibido
    b. Compara con hash de metadata
        Si coincide: crea *ReceivedFile* y lo entrega
        Si difiere: descarta y reporta error

---

# 3. Decisiones de diseño

## 3.1. Modelo de concurrencia adoptado

Se eligió un modelo basado en hilos. La UI corre en el hilo principal del programa; mientras las tareas de red se ejecutan el hilos dedicados. Esto evita el uso de un runtime asíncrono, simplificando el diseño y manteniendo un buen rendimiento.
A su vez, se mantiene un thread de multiplexing que:

- Escucha el socket UDP
- Clasifica cada paquete según su tipo
- Lo enruta a distintos canales de forma no bloqueante

Luego el thread de captura de mensajes, lee directamente desde cada uno de los canales, y realiza el handleo correspondiente.

## 3.2. Gestión de hilos y tareas asíncronas

- Se usa *Arc<Mutex<T>>* para compartir los estados entre hilos (frames, logs, flags, cámara).
- Los hilos de red duermen breves intervalos de tiempo (33 ms) para evitar busy-wait. Al fragmentar en RTP, se puede espaciar micro-sleeps entre paquetes para no saturar la cola del socket.

- *CameraHandler* usa un flag *stop_flag* y *JoinHandle* con drop que hace *join*() para terminar el hilo de captura.
- Los hilos de red consultan flags o salen al detectar errores terminales

## 3.3. Estrategia para el manejo de errores y resiliencia

- Utilización de *Result* y logs con timestamp para diagnosticar errores. Algunos errores repetitivos tienen un límite de logging (p. ej. log cada N fallos de encode) para evitar spam.
- Fallo al crear encoder: log + slee y reintento
- Fallos al codificar un frame: se descarta el frame, se continúa con el siguiente para mantener mantener el tiempo real.
- Paquete RTP fuera de los esperado o incompletos: se descartan para poder continuar con la ejecución
- ICE valida *USERNAME* en Binding Request en el rol Answer, se descarta tráfico de pares no válidos.

## 3.4. Serialización y protocolos de comunicación

- **SDP**: generado como texto plano. Incluye ‘m=video’, ‘rtpmap’, ‘fmtp’ (H.264 con ‘packetization-mode=1’), credenciales ICE, ‘rtcp-mux’. El intercambio es automático a través del signaling server.
- **ICE**: construcción de Binding Request/Success, ‘STUN_COOKIE’, ‘TRANSACTION-ID’, ‘XOR-MAPPED-ADDRESS’. Cuenta con candidatos host y candidatos srflx.
- **RTP**: header v2 con ‘version', ‘marker’, ‘payload type’, ‘sequence’, ‘timestamp’, ‘ssrc’. Serialización/deserialización propia.
- **RTCP**: módulos para SR/RR y métricas básicas; para pérdidas y sincronización RTP.
- **DTSL**: Módulos para generar el handshake DTSL entre los pares remotos, y la inferencia del master_secret
- **SRTP**: módulos para la generación de claves, encriptado y desencriptado de los paquetes RTP para garantizar una transmisión segura.
- **SCTP:** módulo de transmisión de datos a través de Data Channels, mediante serialización y procesamiento de chunks.

## 3.5. Gestión de dependencias y compilación

- 'eframe'/'egui' 0.28: UI de escritorio, 'ColorImage' para texturas.
- 'nokhwa' 0.10 ('input-native', 'output-threaded'): captura de cámara en RGB.
- 'openh264' 0.6: encoder/decoder H.264; conversiones RGB a YUV en el proyecto.
- 'image' 0.24: utilidades de imagen.
- 'rand' 0.9, 'chrono' 0.4: utilitarias.

Se ejecuta en modo ‘--release’ para rendimiento de codificación/decodificación. El ejecutable se genera en ‘target/release/roomrtc’.

## 3.6. Separación entre la UI, signaling y capa multimedia

La UI fue diseñada como una máquina de estados explícita, el Lobby tiene la autenticación y el discovery, el Call contiene la negociación y videollamada, y las transiciones son controladas únicamente por mensajes del servidor o del peer.
Este diseño evita mezclar la lógica de red en la UI, permitir que el render nunca se bloquee, simplificar la depuración del flujo de signaling y alinear el diseño con WebRTC real.

## 3.7. Justificación del modelo de concurrencia

Room RTC utiliza un modelo de concurrencia basado en **hilos del sistema operativo (std::thread)** combinados con **Arc<Mutex<T>>** para compartir estado entre ellos. Esta decisión responde a varios motivos técnicos y pedagógicos relevantes para un proyecto de mediana escala en Rust:

**Simplicidad conceptual y control explícito**

Un modelo basado en hilos evita la complejidad añadida de un runtime asincrónico. Las interacciones entre la UI, captura de cámara, envío y recepción RTP son naturalmente paralelizables bajo el modelo de un hilo por tarea. También permite un control exacto del ciclo de vida de cada worker thread.

**Seguridad directa sin data races**

Se garantiza seguridad de memoria siempre que los datos compartidos están encapsulados en *Arc<Mutex<T>>* y no haya aliasing mutable simultáneo. Elegir este modelo permite que el estado compartido esté protegido por Mutex, la UI pueda leer constantemente estos estados sin bloquearse y los threads puedan actualizarse ellos mismos sus estructuras sin riesgo de corrupción.

**Adecuado para la carga real del proyecto**

El sistema tiene como máximo 4-6 hilos para la UI, la cámara, el envío y recepción e hilos de signaling. Para una aplicación P2P, este modelo es óptimo por la baja latencia y la baja sobrecarga del scheduler, además el código es más simple que un runtime async.

## 3.8. Justificación del diseño ICE-Lite

La implementación de ICE utilizada por Room RTC sigue el enfoque **ICE-Lite**, un modo reducido del estándar ICE (RFC 5245) donde el peer *lite* no realiza *connectivity checks* activos, responde únicamente a **STUN Binding Requests**.
Implementar **ICE-Lite** cumple con los alcances de este proyecto, ya que reduce la complejidad, permite validar el flujo completo ICE/STUN sin implementar toda la RFC y está previsto por la especificación WebRTC.

Las ventajas obtenidas del diseño implementado son:

- Implementación mucho más corta y robusta
- Debugging significativamente más simple

- Delegación del rol controlling al peer que genera la Offer

En conclusión, la elección de ICE-Lite permitió implementar una versión simplificada pero funcional de ICE y completar exitosamente la conectividad P2P.

### 3.9 Jitter Buffer

Se implementa un jitter buffer configurable que suaviza fluctuaciones de red manteniendo una pequeña cola de paquetes RTP ordenados por timestamp y sequence number. Las funciones principales son almacenar paquetes entrantes, descartar paquetes atrasados, reordenar pequeños desvíos y entregar frames al decodificador con ritmo estable. Esto permite mantener una reproducción más fluida del video recibido.

### 3.10 RTCP PLI

Se implementan paquetes PLI en el protocolo RTCP que permiten restaurar la visualización del video recibido cuando se experimenta una tasa alta de pérdida de paquetes que generan fallos en la decodificación de video.
Al detectar errores de decodificación se emite un PLI desde el peer receptor que genera el envío de un keyframe por parte del peer transmisor para restaurar el flujo de frames.

---

## 4. Crates y dependencias externas

El proyecto utiliza diversas librerías externas del ecosistema de Rust (*crates*) para implementar funcionalidades específicas:

### 4.1. Interfaz gráfica y aplicación de escritorio

- eframe y egui: Framework para crear aplicaciones de escritorio multiplataforma con interfaz gráfica declarativa. Eframe proporciona la integración nativa mientras que egui ofrece los componentes de UI de modo inmediato (immediate mode GUI).

### 4.2. Captura de video

- nokhwa: Librería para acceso a dispositivos de cámara web multiplataforma. Permite enumerar cámaras, configurar formatos de captura y obtener frames en diversos formatos de píxel como RGB.

### 4.3. Codificación y decodificación de video

- openh264: Binding de Rust para la librería OpenH264 de Cisco, que proporciona codificación y decodificación H.264/AVC de alta calidad. Esta librería es fundamental para comprimir el video antes de transmitirlo por la red

### 4.4. Captura y reproducción de audio

- cpal: Permite la captura y configuración de parámetros de audio desde el micrófono, así como su reproducción en el par remoto luego de su recepción.

### 4.5. Codificación y decodificación de audio

- opus: Crate que permite manejar el encodeo y desencodeo de el audio capturado por cpal, configurar bitrates específicos, y comprimir el archivo de audio para garantizar una menor latencia durante la transmisión por la red.

### 4.6. Seguridad

- sha2: Funciones hash SHA-256 para generación del fingerprint que se intercambia en el SDP, así como durante el handshake DTSL
- rcgen: Generación de certificados X.509 auto-firmados que se utilizan para asegurar la identidad de los pares durante el handshake DTSL
- aes-gcm: Encriptación AES-128 en modo GCM que se utiliza para el encriptado y desencriptado de los paquetes RTP
- hkdf: HKDF (HMAC-based Key Derivation Function) que se utiliza para derivar el master secret de forma determinística en ambos pares remotos.

### 4.7. Utilidades generales

- rand: Generación de números aleatorios, utilizada para crear identificadores únicos como transaction IDs de STUN, credenciales ICE y valores de sincronización RTP.
- chrono: Manejo de fechas y timestamps, empleada para añadir marcas temporales a los logs de la aplicación.

### 4.8. Networking básico

- if-addrs: Obtención de ip local para la generación de los candidatos host locales.

### 4.9. Protocolos adicionales implementados

- Multiplexado STUN/DTLS/RTP/RTCP/SCTP: para que todos los protocolos de transporte compartan un solo socket UDP.
- DTLS con un handshake simplificado pero funcional para generar claves SRTP y validar identidad.
- SRTP para el cifrado seguro punto-a-punto de los frames de video.
- Protocolo de señalización propio de forma tipo=valor, con fragming con longitud, extensible y tolerante al ruido.

---

## 5. Tecnologías utilizadas

### 5.1. Protocolos de red implementados

**ICE (Interactive Connectivity Establishment) - RFC 5245**

Protocolo encargado de establecer conectividad punto a punto (**P2P**) entre dos clientes, incluso si ambos se encuentran detrás de NATs o firewalls.

Implementación en Room RTC:

- Recolección de candidatos locales (**host y srflx candidates**) desde las interfaces de red del sistema y peticiones al servidor STUN.
- Construcción de pares de conexión entre candidatos locales y remotos con prioridad según tipo de candidato.
- Ejecución de **connectivity checks** mediante el intercambio de mensajes STUN Binding Request/Response entre pares.
- Validación de direcciones accesibles y selección del par de **candidatos óptimo**.
- Establecimiento del canal UDP directo para transmisión RTP/RTCP.

**STUN (Session Traversal Utilities for NAT) - RFC 5389**

Protocolo auxiliar utilizado por ICE para el descubrimiento y verificación de direcciones a través de NATs.

Implementación en Room RTC:

- Implementación parcial del protocolo STUN para descubrimiento de direcciones
- Implementación de mensajes **Binding Success Response** en respuesta a solicitudes válidas.
- Procesamiento del atributo **XOR-MAPPED-ADDRESS** para extraer la IP y puerto visibles desde el peer remoto.
- Validación de respuestas mediante el Magic Cookie (*0x2112A442*).
- Construcción y envío de mensajes **Binding Request** con transaction IDs de 96 bits.

**RTP (Real-time Transport Protocol) - RFC 3550**

Protocolo de transporte en tiempo real utilizado para el streaming de video sobre UDP codificado en H.264.

Implementación en Room RTC:

- Estructura de cabecera RTP de 12 bytes con los campos estándar:
    - Version, Padding, Extension, CSRC Count
    - Marker bit (fin de frame)
    - Payload Type (97 → H.264)
    - Sequence Number (detección de pérdidas y reordenamiento)
    - Timestamp (sincronización temporal)
    - SSRC (identificador de fuente)
- Funciones de serialización y deserialización (*to_bytes() / from_bytes()).*
- Numeración secuencial de paquetes para control de pérdida.

**H.264 RTP Packetization - RFC 6184**

Define cómo se encapsulan las NAL Units del codec H.264 dentro de paquetes RTP. Es esencial para fragmentar frames grandes en varios paquetes sin perder integridad.

Implementación en Room RTC:

- Empaquetado de NAL units H.264 en payloads RTP
- Soporte completo para **Fragmentation Units tipo A (FU-A),** permitiendo dividir NAL Units que superan el tamaño máximo (**MTU ≈ 1200 bytes**).
- Manejo de cabeceras **SPS (Sequence Parameter Set)** y **PPS (Picture Parameter Set**) para inicialización del decodificador remoto.
- **Depacketizer** que reconstruye las unidades de acceso (**Access Units**) antes de la decodificación.

**SDP (Session Description Protocol) - RFC 4566**

Protocolo de descripción de sesión que define las propiedades y parámetros multimedia de la conexión WebRTC.

Implementación en Room RTC:

- Generación de descripciones de sesión "*Offer*" y "*Answer*" mediante *sdp_utils*.
- Inclusión de:
    - Campos *a=ice-ufrag* y *a=ice-pwd* (credenciales ICE)
    - Líneas *a=candidate*: con IP, puerto, prioridad y tipo de transporte
    - Descriptores *m=audio* y *m=video* con codec y protocolo
    - Parámetros de conexión *(IN IP4 ...*)
- Descripción de media streams (audio/video) con codecs y formatos.
- Parseo manual de textos SDP para extraer credenciales y candidatos.
- Implementación del proceso de señalización WebRTC mediante signaling server.

**RTCP (Real-time Transport Control Protocol) — RFC 3550**

Protocolo complementario a RTP que envía información de control, sincronización y estadísticas sobre la calidad de la transmisión.

Implementación en Room RTC:

- Generación y envío periódico (~1 Hz) de **Sender Reports (SR)** y **Receiver Reports (RR).**
- Interpretación de reportes entrantes para mantener sincronización y control básico de calidad.
- Inclusión de los siguientes campos:
    - Conteo de paquetes enviados y bytes transmitidos
    - Marcas de tiempo NTP y RTP asociadas
    - Medición de **jitter**, **pérdidas acumuladas** y **fracción de pérdida**

**DTLS (Datagram Transport Layer Security) - RFC 6347**
Protocolo de handshake seguro que establece autenticación bidireccional entre peers mediante intercambio de certificados X.509 y derivación de secretos criptográficos compartidos.

Implementación en Room RTC:
La implementación en el proyecto difiere a la del RFC, está simplificada. No handlea lo relacionado a paquetes UDP (Retransmisión, Fragmentación, Reordenamiento) y asume que no habrá ese tipo de problemas.

- Al iniciar la aplicación se genera un certificado, y se hashea ese certificado para generar un fingerprint, que será compartido con el par remoto en el SDP
- Cuando se establece la conexión vía ICE, sucede el handshake DTSL, donde los pares intercambian certificados para validar su identidad:
    - Hashean el certificado compartido con el par remoto
    - Validan el resultado del hash contra el fingerprint compartido en el SDP
    - Si coinciden, se establece la conexión, si no coinciden, se rechaza por seguridad.
    - Intercambian valores aleatorios, que se utilizan para generar el master_secret
- Luego del handshake exitoso, los pares derivan deterministicamente el mismo master_secret, sin enviarlo por la red utilizando los valores aleatorios compartidos en el handshake.

**SRTP (Secure Real-time Transport Protocol) - RFC 3711**
Protocolo de encriptación y autenticación para flujos RTP que proporciona confidencialidad del payload, integridad de datos y protección contra replay attacks mediante AES-128-GCM.

Implementación en Room RTC:

La implementación de este protocolo también está simplificada: Se deriva únicamente una key para el encriptado y desencriptado, no se utilizan las auth_key ni salting_key. Tampoco hace validaciones HMAC-SHA1 para garantizar que el contenido del paquete no haya sido modificado por un atacante. Sin embargo la transmisión de los datos ya es segura y no puede ser desencriptada sin las master keys.

- Luego de un handshake DTSL exitoso, se pudo validar la identidad del par remoto, y generar el master_secret.
- Simetría criptográfica: Sender y Receiver derivan la misma clave SRTP a partir del master_secret DTLS.
- Encriptación selectiva: Solo el payload RTP se encripta; el header RTP permanece en plaintext para permitir routing y decodificación en intermediarios

**RTP Payload Format for Opus - RFC 7587**
Protocolo que define la configuración específica del encodeo Opus para garantizar una transmisión de calidad por la red.

Implementación en Room RTC:
Se toman las configuraciones indicadas en el RFC respecto del PT, Clock Rate, Channels, FEC support, minptime y marker bit.

**SCTP (Stream Control Transmission Protocol) - RFC 4960**
Protocolo de transporte confiable para transmisión de datos orientado a mensajes que opera sobre UDP en el contexto de WebRTC Data Channels.

Implementación en Room RTC:
Room RTC utiliza el crate sctp-proto, una implementación pura en Rust,

La librería proporciona una API que se integra con el loop de recepción. El **multiplexor** detecta los paquetes SCTP y los enruta al **DataChannelManager**, que gestiona la asociación SCTP (handshake, streams, timeouts) utilizando **sctp-proto**.

Se implementa el handshake completo de 4 vías (INIT, INIT-ACK, COOKIE-ECHO, COOKIE-ACK), la transmisión confiable de DATA chunks con retransmisiones automáticas y el control de flujo para evitar la congestión de la red.

---

# 6. Metodología de desarrollo

## 6.1. Enfoque metodológico

El desarrollo del proyecto se realizó de manera **iterativa** e **incremental**, con un enfoque ágil adaptado a la escala del equipo. Cada semana se mantuvieron **reuniones** con el tutor de la cátedra, en las cuales se revisó el estado de avance, se establecieron prioridades, se discutieron las tareas pendientes y se definieron los siguientes pasos.
Este esquema permitió mantener un flujo de trabajo continuo, ajustando los objetivos en función del progreso y de los desafíos técnicos que iban surgiendo.

## 6.2. Organización del equipo y roles

El equipo trabajó de forma **colaborativa** y **horizontal**, sin roles formales asignados. Las tareas se distribuyeron semanalmente en función de la disponibilidad y las áreas de conocimiento o interés de cada integrante.
Esta modalidad permitió flexibilidad y una participación equitativa en las distintas etapas del desarrollo (backend, frontend, integración, documentación).

### 6.3. Gestión de versiones y ramas

Para el control de versiones se utilizó Git, con una estrategia basada en GitHub Flow.
Cada nueva funcionalidad o corrección se desarrolló en una rama independiente, que luego era fusionada (merge) a la rama principal (*main*) una vez revisada y verificada su correcta integración.
Este esquema facilitó el trabajo paralelo y redujo conflictos entre los aportes de los distintos integrantes.

### 6.4. Herramientas de colaboración

La **colaboración** y **coordinación** del equipo se realizaron principalmente mediante el **repositorio de GitHub**, utilizando su **pull requests** para revisar y aprobar cambios, acompañado de una comunicación fluida por WhatsApp.
No se emplearon herramientas adicionales de gestión de tareas (como Trello o Jira).

---

## 7. Desafíos y soluciones técnicas

### 7.1. Latencia y pérdida de paquetes

La latencia y la pérdida de paquetes son inherentes a los sistemas de transmisión en tiempo real basados en UDP. El objetivo fue minimizar la latencia perceptible sin sacrificar estabilidad.

**Problemas detectados:**

- Retrasos variables debido al tamaño de los frames H.264 y la fragmentación FU-A.
- Pérdidas ocasionales de paquetes que producían artefactos visuales o cortes en el video.
- Ausencia de control de congestión

**Soluciones implementadas:**

- Se ajustó el tamaño máximo de unidad de transmisión (**MTU = 1200 bytes**) para evitar fragmentación a nivel IP.
- Se empleó el bit **Marker (M=1)** para identificar el fin de cada frame y facilitar la reconstrucción en el receptor.
- Se añadieron **Receiver Reports (RR)** con estadísticas de pérdida y jitter, permitiendo monitorear la calidad de la sesión.
- Se programó el envío de paquetes con micro-pausas proporcionales al tamaño del frame, reduciendo ráfagas que provocan sobrecarga de buffer.
- Se implementó PLI (Picture Loss Indication) para solicitar keyframes inmediatos ante la detección de pérdida de paquetes, permitiendo una recuperación del video.

### 7.2. Compatibilidad entre plataformas

Uno de los principales objetivos de Room RTC fue mantener portabilidad y ejecución nativa multiplataforma, utilizando únicamente dependencias del ecosistema Rust.

**Problemas detectados:**

- Diferencias en el manejo de cámaras entre Windows y Linux (especialmente en máquinas virtuales).
- Necesidad de asegurar que los sockets UDP funcionen correctamente en redes locales con firewalls o NAT simples.

**Soluciones implementadas:**

- Se utilizó la crate nokhwa para acceso multiplataforma a dispositivos de cámara.
- La interfaz gráfica fue desarrollada con *egui/eframe*, que ofrece compatibilidad nativa con OpenGL, garantizando que la aplicación pueda ejecutarse en Windows, Linux y macOS sin modificaciones.
- Para pruebas en entornos virtualizados, se habilitó la opción de dispositivo USB-camera passthrough.

## 7.3. Manejo de reconexiones y errores de red

Debido al uso de UDP sin re-transmisión automática, los errores de conexión o los cortes temporales pueden provocar la interrupción del flujo.

**Problemas detectados:**

- Los peers no detectaban automáticamente la pérdida de conectividad.
- No existía un mecanismo para finalizar la conexión de manera limpia.
- Los sockets quedaban bloqueados si un peer cerraba abruptamente la sesión.

**Soluciones implementadas:**

- Se añadió el botón "**Colgar**" en la interfaz, que cierra las sesiones, libera sockets y detiene los hilos de envío y recepción.
- Se agregaron timeouts y manejo de errores no bloqueantes (*WouldBlock, TimedOut*) para evitar congelamientos en la UI.
- Los sockets UDP se configuran con **read_timeout** (300 ms), permitiendo que el hilo de recepción se mantenga activo y reactivo.

## 7.4. Optimizaciones de rendimiento

El procesamiento de video en tiempo real implica tareas intensivas: captura, codificación H.264, empaquetado RTP y renderizado gráfico.

**Problemas detectados:**

- Evitar bloqueos en la interfaz mientras se codifica video.
- Minimizar la carga de CPU y las copias innecesarias de memoria.
- Mantener una tasa constante de 30 fps con codificación en software.

**Soluciones implementadas:**

- Uso de multithreading mediante *std::thread::spawn* para aislar las tareas de captura, envío y recepción.
- Compartición de recursos mediante *Arc<Mutex<T>>*, garantizando seguridad en concurrencia sin bloqueos largos.
- Codificación y decodificación **asíncrona** con buffers temporales, desacoplando los tiempos de captura y transmisión.
- Control de tasa de envío: cada frame se envía con pausas de ~42 ms, adaptándose al framerate.
- Registro detallado de tiempos (*chrono* + *logs*) para identificar cuellos de botella.

---

# 8. Conclusiones y trabajo futuro

## 8.1. Conclusiones generales

Room RTC implementa de manera integral los componentes fundamentales de un stack WebRTC real, abarcando los siguientes puntos:

- Señalización mediante servidor TCP
- Negociación SDP completa
- Establecimiento de conectividad P2P con ICE y STUN
- CIfrado mediante DTLS simplificado y derivación de claves SRTP
- Transmisión de video H.264 con paquetización RTP/RTCP
- Tranferencia de archivos mediante SCTP Data Channels
- Interfaz gráfica funcional para discovery y videollamada

El proyecto demuestra la viabilidad de construir un sistema de videoconferencia P2P desde cero utilizando únicamente **Rust** estándar y librerías de propósitos generales. La modularidad alcanzada permite comprender cada capa del stack WebRTC a bajo nivel y facilita futuras extensiones sin reescrituras profundas.

## 8.2. Lecciones aprendidas

La implementación manual de los RFCs clave (ICE, STUN, RTP, H.264 y RTCP) demanda atención a los detalles de formato y temporización. Un diseño modular y pruebas por componente facilitan el desarrollo incremental. La elección de hilos y sincronización simple resultó adecuada para un prototipo robusto y comprensible.