# Taller de programación

# Entrega intermedia - 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.

---

# 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, lo cual desencadena la inicialización de componentes internos como el módulo de la cámara, los agentes **ICE** y la generación del **SDP** local. A partir de este punto, se puede establecer una conexión con un peer remoto y 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. Actualmente, en **Room RTC** esta señalización se realiza manualmente, en donde el usuario copia el "*SDP Offer*" creado de un peer y lo pega en el otro peer para poder generar la "*SDP Answer*", simulando el intercambio que a futuro realizará un servidor de *señalización*.

- 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.
Periódicamente, se envían y reciben *Sender Reports (SR)* y *Receiver Reports (RR)*, que calculan jitter, pérdida de paquete y sincronización temporal.

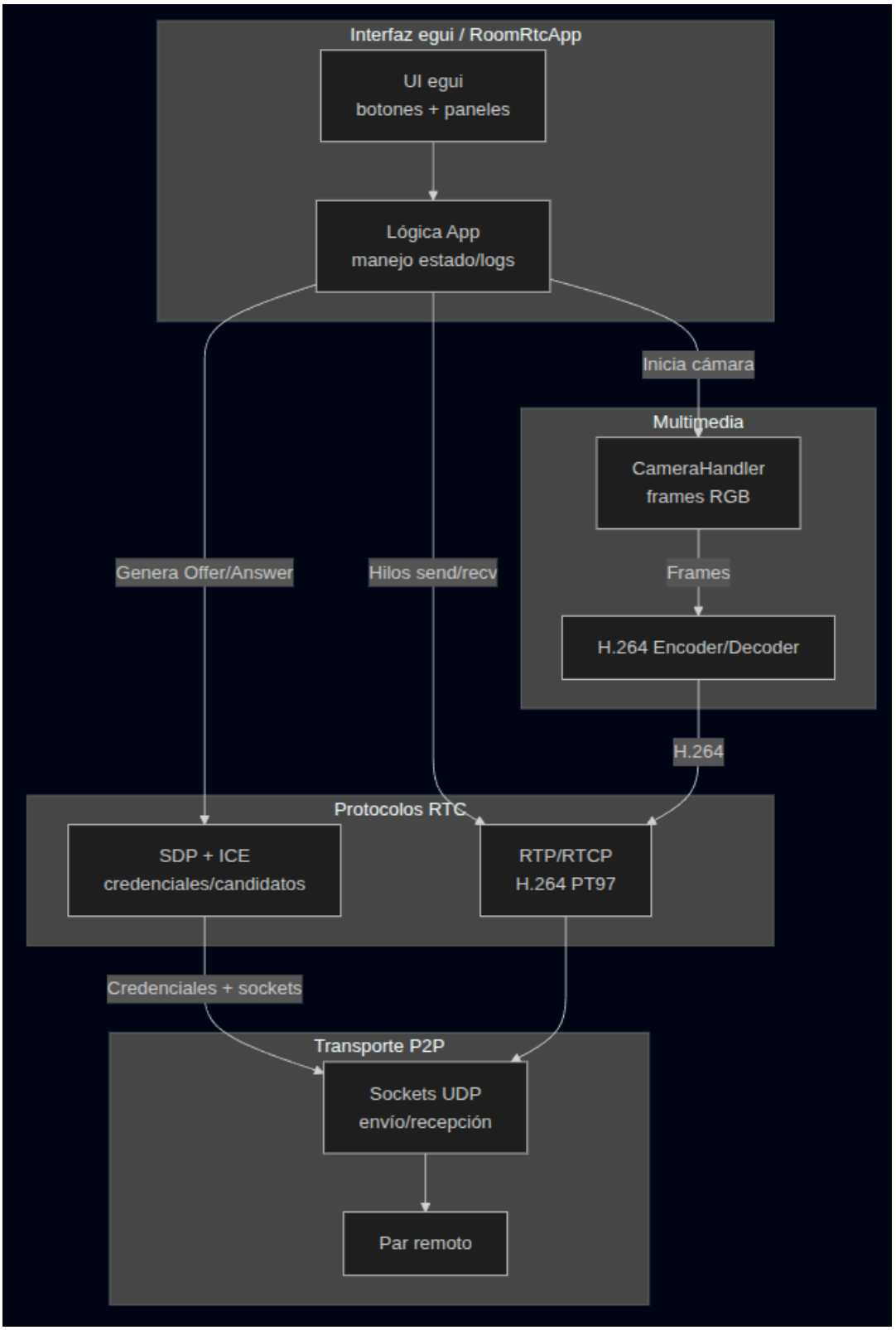
- Módulo multimedia
  Utilizando el crate **nokhaw** para capturar frames en formato RGB se abstrae el acceso al dispositivo de cámara. Estos frames son comprimidos en formato H.264 utilizando bindings de **OpenH264** y luego paquetizados. El peer remoto recibe los paquetes RTP, los construye y decodifica, generando un *ColorImage* que se renderiza en pantalla.

- 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 autenticación y gestión de usuarios
  Actualmente, Room RTC no implementa autenticación ni gestión de usuarios persistentes, ya que el objetivo de esta entrega era concentrarse en la capa multimedia y de transporte. Sin embargo, el intercambio de credenciales **ICE** cumple el rol equivalente de autenticación temporal durante la negociación **P2P**.

## 2.3. Diagrama de arquitectura

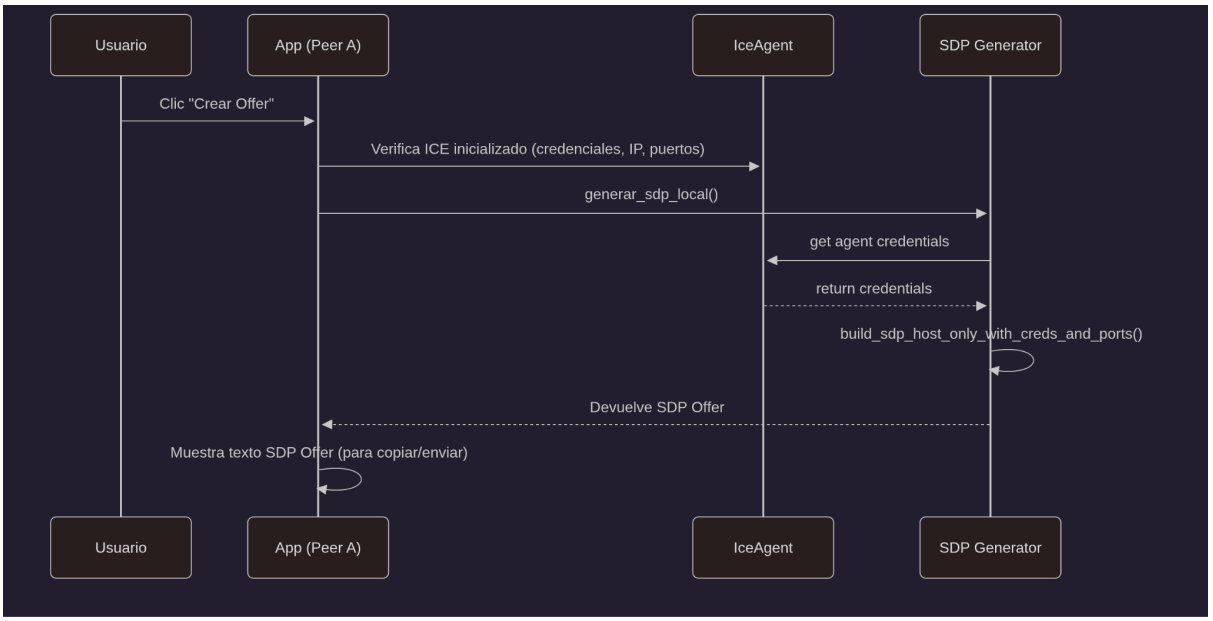



El diagrama muestra la organización modular del sistema, donde la interfaz gráfica (*egui/eframe*) interactúa con la lógica central para gestionar cámara, estado y logs. Los frames capturados por *CameraHandler* se codifican en **H.264**, se transmiten mediante los

protocolos **RTP/RTCP** e **ICE** sobre sockets **UDP**, y finalmente se decodifican y renderizan en el peer remoto, manteniendo una separación clara entre interfaz, multimedia, protocolos y transporte.

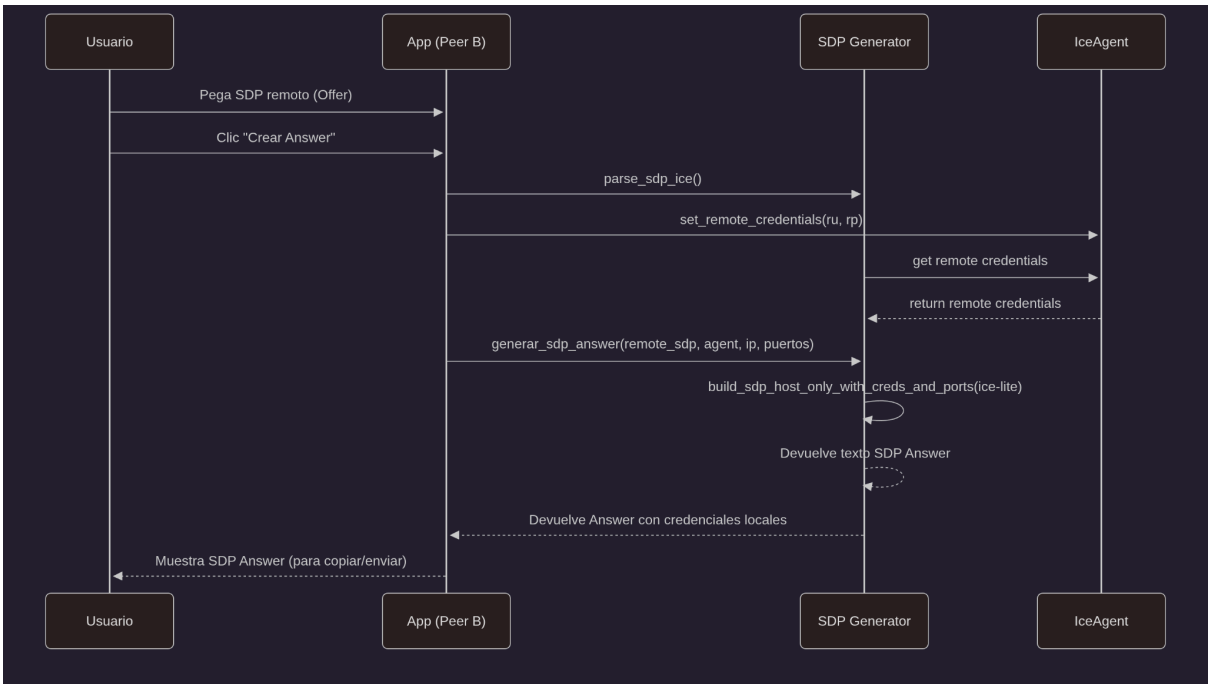
## 2.4. Diagramas de secuencia

### 2.4.1. SDP Offer



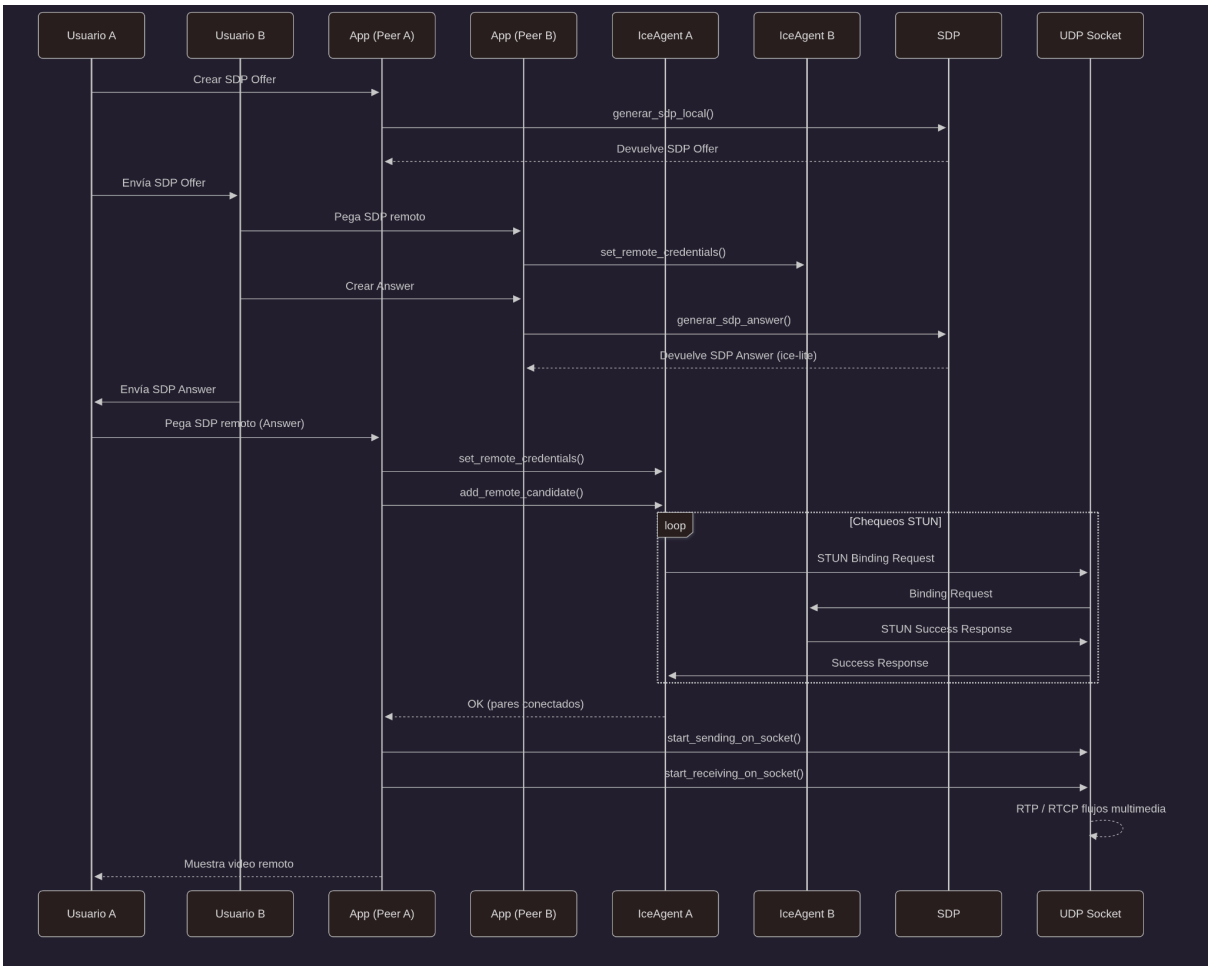

El diagrama muestra el flujo de mensajes entre el usuario, la aplicación y los módulos **ICE** y **SDP** durante la creación de una oferta de sesión.
Al presionar "*Crear Offer*", la aplicación valida la inicialización del agente **ICE** y ejecuta *generar_sdp_local(),* que obtiene las credenciales y construye la descripción de sesión con *build_sdp_host_only_with_creds_and_ports().*
El texto **SDP** resultante se devuelve a la interfaz y se presenta al usuario para su envío al peer remoto.

### 2.4.2. SDP Answer



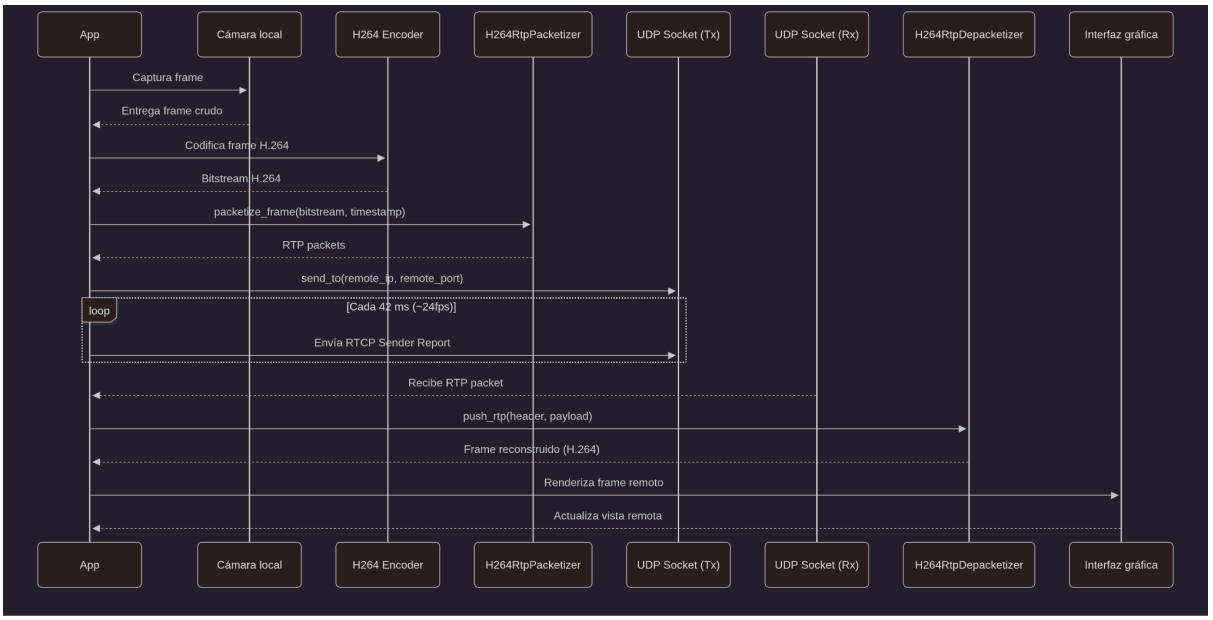

El diagrama muestra el proceso en el que el **Peer B** recibe una *SDP Offer*, extrae las credenciales **ICE** remotas y genera una *SDP Answer* en modo **ICE-Lite**.
La aplicación analiza la *Offer* con *parse_sdp_ice(),* configura las credenciales con *set_remote_credentials(),* y construye la *Answer* mediante *generar_sdp_answer().*
El resultado se muestra en la interfaz para ser compartido con el **Peer A**.

### 2.4.3. Conexión entre pares



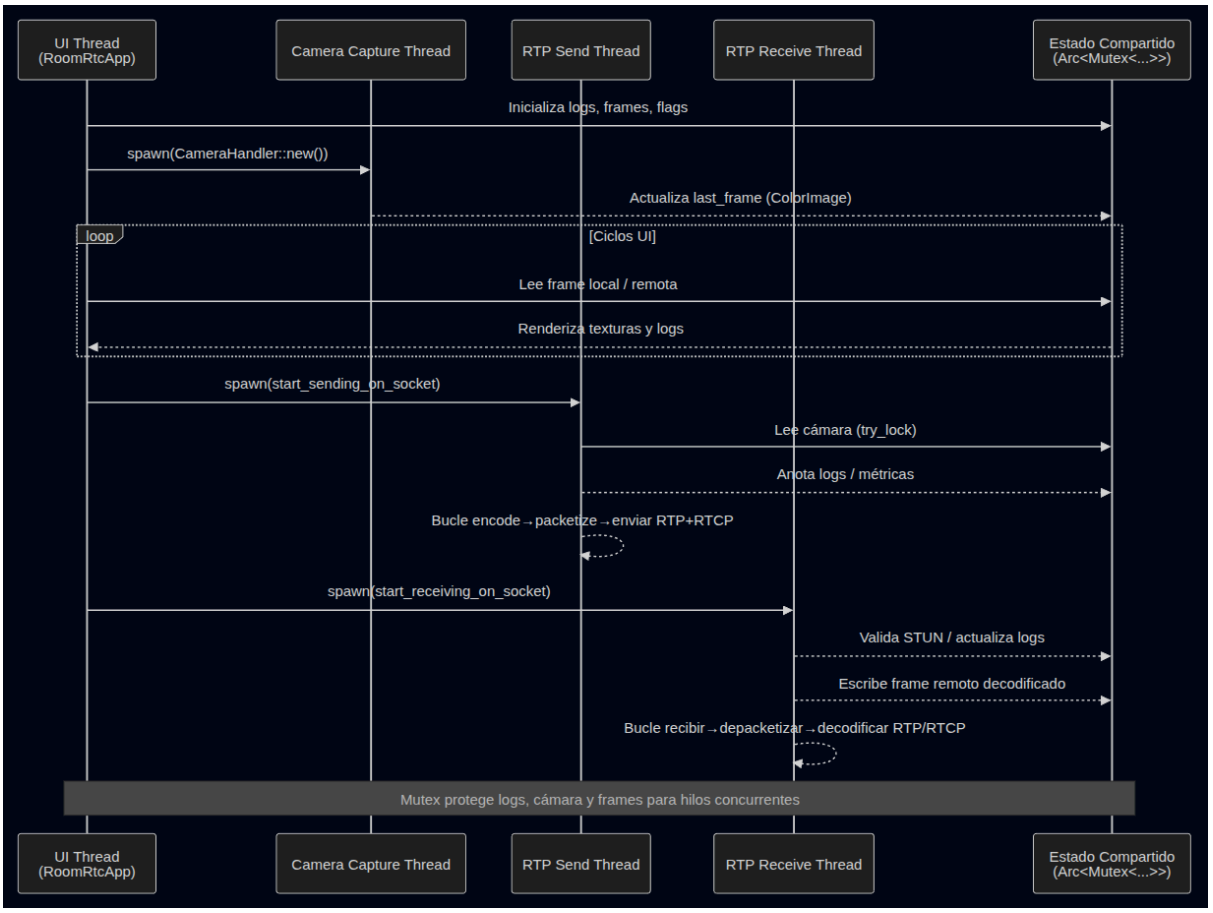

Secuencia completa del establecimiento de conexión WebRTC manual entre dos peers: generación de *Offer/Answer*, intercambio manual de **SDP**, configuración de credenciales **ICE**, checks **STUN** hasta formar el par **P2P**, y posterior inicio de transmisión **RTP/RTCP** directo por **UDP**.

### 2.4.4. Transmisión de video




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** y enviados por **UDP**.
En el extremo remoto, los paquetes **RTP** se re ensamblan y decodifican, reconstruyendo el video que se actualiza en la interfaz gráfica aproximadamente cada 42 ms (*~24 fps*), con control periódico mediante **RTCP Sender Reports**.

### 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.

## 2.5. Flujo de comunicación entre peer y peer

1. **Inicialización local**:
   El peer detecta su IP y asigna puertos libres (audio y video), y se generan los candidatos ICE y credenciales (*ufrag*, *pwd*).
2. **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.
3. **Comprobación ICE (STUN):**
   Ambos peers intercambian *Binding Request* 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 el cliente que genera la answer toma

el rol de cliente ICE-Lite que implementa respuesta para STUN binding request pero no genera consultas propias.

4. **Transmisión multimedia:**
   Se capturan frames locales, se codifican, se fragmentan en RTP y se envían al peer remoto. En paralelo, el receptor depaquetiza, decodifica y renderiza el video recibido.
5. **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.

---

# 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.

## 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 manual (copiar/pegar).
- **ICE**: construcción de Binding Request/Success, ‘STUN_COOKIE’, ‘TRANSACTION-ID’, ‘XOR-MAPPED-ADDRESS’. Solo candidatos ‘host’.
- **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.

## 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'.

---

# 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. 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.

## 4.5. Networking básico

El proyecto utiliza exclusivamente la biblioteca estándar de Rust (*std::net*) para todo el manejo de sockets UDP, sin dependencias externas adicionales para networking. Esto incluye:

- *UdpSocket*: Para envío y recepción de paquetes
- *IpAddr, SocketAddr*: Para manejo de direcciones IP y puertos

---

# 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 candidates**) desde las interfaces de red del sistema.
- Construcción de pares de conexión entre candidatos locales y remotos.
- 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.
- Simulación del proceso de señalización WebRTC (intercambio manual de SDP entre peers).

**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**

---

# 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.

## 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

Se implementan los componentes esenciales para la transmisión de video P2P: generación de SDP, establecimiento de conectividad con ICE/STUN, transporte RTP con paquetización H.264 y control RTCP.

La UI permite operar el flujo manualmente y visualizar el video local/remoto en tiempo real

## 8.2. Limitaciones del proyecto

- Solo candidatos ‘host’ (no existe STUN server ni TURN)
- Sin DTLS/SRTP: no hay cifrado de paquetes
- Solo video mediante H.264
- Signaling manual (copiar/pegar el SDP)
- No sistema de configuración mediante archivo. Actualmente hay parámetros que se encuentran hardcodeados.

## 8.3. Próximas mejoras o extensiones

- Integrar un STUN server para permitir la conectividad fuera de entornos locales.
- Implementar cifrado end to end.
- Negociación de codecs.
- Aumento de coverage mediante tests.
- Uso de datos transmitidos por RTCP para modificar dinámicamente la calidad de transmisión de video.

## 8.4. 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.