Plataforma de Videoconferencia basada en WebRTC "RoomRTC"

Implementación en Rust

Índice

1. Introducción

El presente documento tiene como objetivo describir el desarrollo e implementación de una plataforma de videoconferencia basada en WebRTC, compuesta por dos módulos principales: una API WebRTC y una aplicación cliente denominada RoomRTC.

El proyecto surge con el propósito de explorar y comprender los mecanismos internos de comunicación en tiempo real sobre redes IP, tales como el intercambio de descripciones de sesión (SDP), la negociación de candidatos ICE (Interactive Connectivity Establishment) y la transmisión de medios mediante los protocolos RTP/RTCP. A diferencia de otras soluciones que emplean bibliotecas de alto nivel, este trabajo implementa gran parte del flujo de comunicación desde cero, con el fin de profundizar en los principios de diseño y en la gestión de bajo nivel del video y el audio.

1.1. Arquitectura general

Módulo WebRTC: Implementa la lógica de señalización y comunicación entre pares. Gestiona la creación de sesiones, el intercambio de mensajes SDP, la negociación ICE y el control de estados de conexión. Esta API actúa como puente entre los clientes, permitiendo establecer canales para la transmisión multimedia.

Módulo RoomRTC: es la aplicación de usuario final que utiliza la API para crear y unirse a salas de videoconferencia. Permite la captura de video desde cámaras locales, el envío de flujos multimedia codificados en H.264, y la recepción y decodificación de video remoto. Además, ofrece una interfaz visual similar a la de plataformas modernas de videollamadas.

Módulo Client: Este módulo encapsula toda la lógica necesaria para que un usuario se conecte a una sala de comunicación, interactúe con otros participantes y gestione los flujos de medios (audio/video). Interactúa estrechamente con el módulo ui en screens y screen_manager para presentar la interfaz gráfica y con el módulo config para obtener los parámetros de funcionamiento.

1.2. Objetivos

El objetivo principal del presente Trabajo Práctico es desarrollar una versión en el lenguaje Rust del stack WebRTC, de manera tal que sea posible la realización de videoconferencias entre usuarios en distintos dispositivos. Los requerimientos funcionales presentes son:

WebRTC:

- **SDP (Session Description Protocol):** Utilizado para establecer la conexión entre pares.
- **ICE (Interactive Connectivity Establishment):** Protocolo de comunicación utilizado para definir la ruta por la que se enviará el video.
- **RTP (real-time transport protocol) y RTCP (RTP control protocol):** Protocolo para realizar la transmisión de video.

Signaling Server: Encargado del discovery de peers. En este caso, el servidor central actuará de signaling server.

2. Protocolos

A continuación, se describen los principales protocolos utilizados en la implementación del proyecto:

2.1. SDP (Session Description Protocol)

El protocolo SDP se utiliza para describir las características de una sesión multimedia. Permite que los pares intercambien información sobre los códecs, direcciones, puertos y otros parámetros necesarios para establecer la comunicación. En esta implementación, cada componente está modularizado: Los valores atómicos y particulares en: `address_type.rs`, `media_type.rs`, `net_type.rs`, `transport_protocol.rs`, `property_attribute.rs`. Se combinan para crear las líneas completas del SDP en: `origin.rs`, `time.rs`, `media_description.rs`, `attribute.rs`, `value_attribute.rs`. Finalmente, la estructura principal que une todo y representa un SDP completo es: `session_description.rs`.

Estructura de `SessionDescription`:

```
pub struct SessionDescription {
  version: SdpVersion,
  origin: Origin,
  time: Time,
  media_description: Vec<MediaDescription>,
  attributes: Vec<Attribute>,
}
```

2.2. ICE (Interactive Connectivity Establishment)

El protocolo ICE es el procedimiento estándar para establecer conectividad P2P a través de NATs y firewalls; descubre rutas posibles entre dos endpoints y elige la que funcione mejor. En esta implementación se separa en componentes simples: `candidate.rs` define el tipo de candidato (host y reflexivo), `pair.rs` modela los pares y su estado (Waiting, InProgress, Succeeded, Failed) y `agent.rs` ejecuta la lógica. El `IceAgent` genera credenciales (ufrag/password), registra candidatos host reutilizando el socket local y consulta un servidor STUN para obtener candidatos server-reflexive. Luego arma todos los pares posibles, los ordena por prioridad y realiza connectivity checks mediante Binding Requests/Responses STUN hasta seleccionar un par válido. El módulo `sdp_helper` convierte esas credenciales y candidatos a líneas SDP para ser intercambiadas por señalización.

3. Stack WebRTC (room_rtc)

El crate `room_rtc` concentra la lógica de WebRTC "bajo nivel": sockets UDP con soporte STUN, agente ICE, construcción/parsing de SDP, envío de RTP/RTCP, cifrado liviano SRTP y un pipeline multimedia con OpenCV y OpenH264.

3.1. PeerConnection e ICE

`RtcPeerConnection` envuelve al `IceAgent` y a un `PeerSocket` (UDP). Expone las operaciones propias del rol:

- `create_offer()`: sólo disponible para el rol **controlling**, genera la descripción local (SessionDescription) a partir de los candidatos registrados y credenciales ICE.
- `process_offer()`: usado por el rol **controlled**; parsea la oferta SDP remota, registra credenciales y candidatos, y devuelve una respuesta SDP.
- `set_remote_description()`: aplica la respuesta cuando somos controlling.
- `start_connectivity_checks()`: inicia los STUN Binding sobre los pares generados y, al hallar un par válido, registra la dirección remota en el `PeerSocket`.

El `PeerSocket` responde automáticamente Binding Requests entrantes y enruta todo lo demás por un canal a los workers multimedia. De esta forma se asegura que ambos peers puedan descubrirse aunque haya NAT y que el socket quede apuntado al par ICE elegido.

3.2. RTP/RTCP y SRTP

`RtcRtpSender` toma NALUs H.264 y arma paquetes RTP (con fragmentación FU-A cuando el NALU es mayor a MTU). Usa un SSRC fijo (1000), incrementa sequence/timestamp a 90 kHz y actualiza métricas de envío. Si las aplicaciones intercambian una clave base64 por señalización, se crea un `SrtpContext` que cifra el payload RTP mediante un keystream XOR derivado de seq/timestamp + clave. `RtpReceiverThread` desencripta (si hay clave), detecta RTCP, actualiza métricas, pasa los RTP por un jitter buffer y reconstruye frames en orden. `RtcpReporterThread` envía periódicamente SenderReport/-ReceiverReport con estadísticas de transmisión/recepción.

3.3. Pipeline multimedia y threads

`WorkerMedia` orquesta todos los hilos necesarios para la videollamada:

- **CameraThread** lee frames de la cámara (`camera_opencv.rs` selecciona backend V4L2/GStreamer o MSMF/DSHOW con fallback de resolución).
- **EncoderThread** convierte RGB→YUV y codifica con OpenH264.
- **RtpSenderThread** envía los bytes H.264 por RTP (con SRTP opcional).
- **RtpReceiverThread** recibe desde el socket, desarma RTP/RTCP, usa jitter buffer y pasa el flujo decodificado.
- **DecodeThread** decodifica H.264 a BGR para previsualización/remoto.
- **RtcpReporterThread** publica SR/RR cada segundo.

La estructura expone canales para obtener la previsualización local, los frames remotos, un sender para inyectar bytes entrantes y un `Arc<Mutex<MediaMetrics` compartido.

3.4. Métricas de llamada

`MediaMetrics` calcula bitrate, jitter, pérdida de paquetes, `highest_seq` y fracción perdida basándose en RTP/RTCP recibidos. **VideoCall** muestra esos valores en UI y marca la conexión como inestable si el tiempo desde el último paquete supera 2 s; si pasan 20 s sin tráfico, se asume desconexión y se corta la llamada enviando RTCP BYE o un mensaje `CALL_END`.

4. Servidor de señalización

El binario `signaling_server` (`src/server/server.rs`) es el encargado del discovery y relay de SDP/ICE. Levanta un `TcpListener` en `0.0.0.0:8443` con TLS (rustls + certificado self-signed generado con rcgen) y maneja un hilo por cliente. `ServerState` mantiene usuarios cargados desde `users.txt`, clientes conectados, estados (Disconnected/Available/Busy) y el mapa de llamadas activas.

4.1. Flujo y estados

Al conectarse un cliente, `handle_client` crea un canal de salida y espera mensajes de texto. `REGISTER` persiste el usuario en archivo plano; `LOGIN` autentica y marca AVAILABLE difundiendo `USER_STATUS_CHANGED`. `CALL_OFFER` verifica que el destino exista y esté libre, marca a ambos como Busy y avisa con `INCOMING_CALL`; `CALL_ANSWER` puede aceptar (`CALL_ACCEPTED` con SDP y clave SRTP) o rechazar (`CALL_REJECTED`) liberando estados. `CALL_END` y desconexiones limpian `active_calls`, devuelven AVAILABLE y notifican `CALL_ENDED` al peer. `LOGOUT` cierra sesión y marca DISCONNECTED. Todos los cambios de estado se broadcastean a los clientes conectados.

4.2. Protocolo de mensajes

Formato de una línea: `TIPO|campo:valor|campo:valor\n`. No se usa JSON para evitar dependencias. La conexión va cifrada con TLS; el cliente confía en el certificado self-signed (verificador inseguro). Mensajes relevantes: `REGISTER`, `LOGIN`, `GET_USERS`, `CALL_OFFER`, `CALL_ANSWER` (accept:true/false), `CALL_REJECT`, `CALL_END`, `ICE_CANDIDATE`, `LOGOUT`. `USER_LIST` devuelve `username:STATUS` por cada usuario registrado; `ICE_CANDIDATE` simplemente relaya la cadena escapada.

5. Cliente RoomRTC

5.1. Cliente de señalización y P2P

`SignalingClient` abre un `TcpStream` TLS con verificador inseguro, escapa payloads (SDP/ICE), procesa cada línea y la mapea a eventos (`UserList`, `IncomingCall`, `CallAccepted`, etc.) para la UI. `P2PClient` envuelve `RtcPeerConnection` y `WorkerMedia`: puede crear/aceptar ofertas, iniciar ICE, enviar/recibir mensajes, setear clave SRTP y detectar un RTCP BYE como fin de llamada. El trait `WebRTCHandler` comparte la lógica entre **WaitingCall** (quien llama) y **JoinMeet** (quien recibe), evitando duplicar `create_offer/process_offer/st`

5.2. Flujo de pantallas

MainApp funciona como un state machine sobre egui:

- **Login:** permite registrar y loguear; guarda mensajes de estado y arranca la sesión de señalización con la dirección del config.
- **Lobby:** muestra usuarios y estados; botón de "VideoCall" inicia **WaitingCall**; **Log Off** manda `LOGOUT` y vuelve al login.
- **WaitingCall (caller):** genera oferta SDP + clave SRTP, la envía con `CALL_OFFER` y espera `CALL_ACCEPTED/REJECTED`; permite saltar a video cuando ICE está iniciado.
- **JoinMeet (callee):** recibe `INCOMING_CALL`, permite aceptar/rechazar, genera answer SDP, setea clave SRTP, inicia ICE y luego cede el cliente a **VideoCall**.
- **VideoCall:** arranca la cámara cuando hay ICE activo, muestra preview local/remota, estadísticas de calidad y permite colgar (RTCP BYE + `CALL_END`). Si detecta silencio prolongado del canal, corta y vuelve a Lobby.

6. Configuración y ejecución

Los parámetros se leen desde archivos plano `clave=valor`.

- `server.conf`: `server_addr` (por defecto `0.0.0.0:8443`), `users_file` (`users.txt`), `log_file` y `max_clients`.
- `client.conf`: `server_addr`, `users_file`, `max_clients`, `log_file` y parámetros de video (ancho, alto, fps). Si no existe se usan defaults.

Ejecución típica:

1. **Servidor:** `cargo run --bin signaling_server -- server.conf` (crea `users.txt` si no existe y genera el cert TLS en caliente).
2. **Cliente GUI:** `cargo run --bin roomrtc -- client.conf` (abre la ventana de RoomRTC, se conecta al servidor y muestra el lobby).

7. Pruebas y validación

El crate `room_rtc` incluye tests unitarios para ICE (prioridad de candidatos, pares, gathering STUN), SDPICE en `RtcPeerConnection` y roundtrip de SRTP. Manualmente se verifica el flujo con dos clientes: registrar/login, obtener `USER_LIST`, iniciar llamada, aceptar, intercambiar ICE, ver video bidireccional y colgar (`CALL_END` o BYE). La UI muestra bitrate, jitter y pérdida; además se monitorea `roomrtc.log` para revisar eventos de señalización y errores de cámara o encoding.

8. Investigación y conclusiones

Se exploraron tres frentes principales:

- **Señalización mínima para WebRTC:** formato texto sin dependencias, TLS con certificado self-signed y verificador inseguro en cliente. *Conclusión*: viable para laboratorio, pero requiere control estricto de estados para evitar duplicación de sesiones o llamadas.
- **ICE/SDP de bajo nivel:** construcción manual de candidatos y prioridades; reuso de socket para host + srflx y ordenamiento de pares mejoró el tiempo de conexión. *Conclusión*: los timeouts STUN y la persistencia del listener son críticos para evitar cierres de canal.
- **Pipeline multimedia:** OpenCV + OpenH264 + hilos dedicados (captura, encode, RTP/RTCP, jitter buffer, decode). *Conclusión*: el diseño multihilo mantiene latencias bajas, pero exige métricas (RTCP) para detectar desconexiones o inestabilidad.

Conclusión general: la pila completa funciona end-to-end (ICE → DTLS → SRTP + video), pero para producción habría que reemplazar el SRTP ligero por un perfil estándar y endurecer la gestión de certificados.

9. Reglas de negocio y decisiones

Reglas de negocio implementadas:

- Un usuario no puede iniciar sesión dos veces; `LOGIN` falla si ya está conectado.
- Estados de presencia: `DISCONNECTED/AVAILABLE/BUSY`; sólo se permite llamar a `AVAILABLE`.
- Una llamada marca `BUSY` a ambos hasta `CALL_END`, rechazo o desconexión.
- Señalización siempre por TLS; el cliente acepta certificado self-signed.

Decisiones técnicas clave:

- Protocolo de señalización plano para depurar rápido.
- Fingerprint DTLS obligatorio en SDP; handshake exporta clave SRTP.
- SRTP casero (XOR) para demos rápidas; punto de mejora futura hacia AES/HMAC.
- Persistencia en archivo plano (`users.txt`) para simplificar despliegue y pruebas.
- UI en egui/eframe para portabilidad y velocidad de iteración.

10. Comunicación entre threads

Hilos presentes y rol:

- **UI (eframe):** hilo principal que pinta egui y coordina acciones; invoca SignalingClient y P2PClient.
- **SignalingClient loop:** `run_client_loop` en hilo propio, lee TLS, mapea líneas a `SignalingEvent` y las envía por mpsc a la UI; `flush_outgoing` vacía el canal de salida.
- **Hilo de conexión P2P:** lanzado en `establish_connection`, corre checks ICE y handshake DTLS.
- **Listener P2P:** hilo que consume el receiver de `PeerSocket`, intenta SRTP unprotect y encola RTP/RTCP hacia WorkerMedia o mensajes a la UI.
- **Listener PeerSocket:** hilo interno de `PeerSocket::listener` que demultiplexa STUN/DTLS/RTP y envía DTLS a su canal y el resto a P2P.
- **WorkerMedia:**
  - CameraThread: captura frames.
  - EncoderThread: codifica H.264.
  - RtpSenderThread: arma y envía RTP/SRTP.
  - RtpReceiverThread: recibe bytes, detecta RTCP, descifra SRTP y usa jitter buffer.
  - DecodeThread: decodifica H.264 a BGR.
  - RtcpReporterThread: envía SR/RR periódicamente.
- **Servidor:** `main` acepta conexiones; `handle_client` corre en un hilo por cliente y maneja señalización.