SHAREPLAY

Trabajo final de grado - DAM

Carlos Mateo Esteban y Juan Pujalte Martínez

Contenido

1. Introducción.

Se ha llevado a cabo una aplicación de escritorio que nos permite a los usuarios crear o unirse a salas virtuales con el propósito de visualizar el contenido multimedia local de forma sincronizada entre los integrantes de la misma sala. La aplicación sé que es de instalación local, tiene integrado el reproductor ya desarrollado y libre MPV, que es un reproductor de código abierto y ligero que es compatible con múltiples formatos y una amplia documentación libre para ser controlado mediante código.

Cada usuario seleccionara de forma local el archivo a reproducir ya sea pasandole al programa una ruta antes de iniciar el reproductor o añadiéndolo después de iniciarlo. Una vez se inicia la reproducción, todos los reproductores de los usuarios de la sala virtual se sincronizan a través de una comunicación con un servidor central que consigue una visualización simultanea entre los integrantes. La sincronización incluye las funciones de pausa, reanudación y ajuste del tiempo de reproducción para los integrantes de la sala que asegura que todos estén viendo el contenido en el mismo punto.

El objetivo principal de la aplicación es conseguir una experiencia de visualización multimedia compartida y de forma remota, manteniendo los reproductores sincronizados entre los integrantes de la misma sala sin necesidad de compartir los archivos, reproducirá y sincronizara todo lo que se tiene como usuario de forma local.

La interfaz de usuario se ha diseñado para que sea y tenga un aspecto intuitivo que permita a los usuarios conectarse a una sala mediante la introducción de una serie de datos. Al presionar el botón de inicio se lanza la reproducción del video y se establece una sincronización entre todos los integrantes de la misma sala.

Para lograr las funcionalidades esperadas se ha implementado una arquitectura cliente-servidor. El servidor funciona como intermediario de todas las partes, gestionando las salas activas y coordinando los reproductores y los eventos que estos de cada reproductor local entre los demás integrantes. Cada solicitud de los usuarios se maneja en hilos independientes con el fin de conseguir una gestión concurrente y evitando bloqueos de comunicaciones.

Para las tecnologías utilizadas vamos a nombrarlas brevemente para finalmente desarrollarlas y justificar su utilización más adelante, habiendo usado las siguientes: Kotlin, JavaFX, IntelliJIDEA, MPVPlayer, Sockets TCP/IP, Multithreading un control de versiones con GIT.

La colaboración entre los desarrolladores se basa en un uso de herramientas de control de versiones y una planificación y división de las tareas de manera de que se lleve a cabo de una forma más clara y controlada. Se estableció un flujo de trabajo que se basa en ramas locales que posteriormente se van mergueando al repositorio principal a medida que se va llevando a cabo el desarrollo de la aplicación y controla la resolución de conflictos de una forma controlada.

2. Análisis.

Introducción.

Para el desarrollo de la aplicación ha sido necesario realizar un análisis detallado de todos los requisitos funcionales y técnicos que esta necesita. El análisis nos ha permitido establecer una base sólida para una posterior implementación que nos garantiza que cada componente cumpla con el propósito esperado y que el sistema funciona de una forma eficiente y sea escalable.

Además, se ha llevado a cabo un trato entre los desarrolladores para organizar el trabajo con una forma colaborativa y estructurada. Se ha utilizado GitHub como plataforma de control de versiones lo que ha permitido mantener un historial claro de los cambios y todos los mergeos llevados a cabo por cada usuario hacia la rama principal asegurando una integración continua del desarrollo.

Para los requisitos técnicos de y de arquitectura de la aplicación ha supuesto uno de los principales retos la gestión de múltiples usuarios conectados y controlados simultáneamente en el que se ha desarrollado una arquitectura basada en un modelo cliente-servidor multihilo:

- Cada cliente representa un hilo de ejecución independiente dentro del servidor.
- El servidor debe de ser capaz de aceptar múltiples conexiones concurrentes y procesar dichas conexiones y posteriores eventos en tiempo real manteniendo sincronizadas las salas activas.
- Cada sala funciona como una instancia lógica donde los eventos de reproducción que hemos descrito se envían a todos los clientes mediante un sistema broadcast.

Este enfoque requiere una gestión eficiente y funcional de la concurrencia, tanto en el servidor como en los clientes. En el servidor se ha usado hilos para manejar cada conexión de los clientes entrantes y estructuras de datos que se sincronizan para gestionar el estado de cada sala y sus integrantes. En el cliente se ha implementado un modelo de paralelización de procesos:

- Escuchar eventos enviados por el servidor.
- Controlar el reproductor MPV mediante comandos.
- Detectar cambios locales en el reproductor.
- Enviar eventos al servidor en tiempo real tras detectar cambios.

La comunicación entre los clientes y el servidor se ha llevado a cabo la implementación de sockets TCP/IP, lo que permite una transmisión de datos fiable, segura y en tiempo real. Para gestionar la concurrencia en Kotlin, se han utilizado coroutines, se trata de una herramienta nativa de Kotlin que nos permite manejar múltiples tareas de manera asíncrona para una eficiente y menor sobrecarga de los hilos tradicionales ya que se tratan de hilos virtuales.

En el servidor, cada cliente se gestiona en un hilo virtual independiente y se ha diseñado un sistema de eventos que nos permite enviar e informar de los cambios de estado a todos los miembros de la sala. Este sistema garantiza que todos los reproductores estén sincronizados, que incluso cuando un nuevo usuario pueda unirse a una sala ya activa.

El cliente deberá de tener múltiples responsabilidades que deben ejecutarse de manera paralela o concurrente:

- Conectarse al servidor y mantener la conexión activa.
- Escuchar eventos de sincronización y aplicarlos al reproductor MPV de manera efectiva.
- Debe detectar eventos locales ya descritos y notificarlos al servidor.
- Debe de poder detectar que el evento que recibe lo ha recibido por primera vez y no es el evento que los demás clientes le mandan después de mandar el mismo un evento, para no causar un mensaje exponencial.
- Gestionar la interfaz gráfica y las interacciones del usuario.

Para controlar el reproductor MPV, se ha pensado en la interfaz JSON IPC, que nos permitirá enviar comandos y leer eventos desde procesos externos. Esto ha facilitado la integración con Kotilin y la sincronización efectiva del estado de los reproductores locales.

Para la interfaz gráfica se ha pensado en el desarrollo con JavaFX, para un desarrollo de ventanas sencillas y que no redimensionaremos dado que contine los campos y la información justos y el control y principal gestión será en el reproductor que si lo será. Los campos necesarios para conectarse deberán de ser la dirección de servidor, el nickname y el nombre de la sala y como forma opcional y como hemos comentado la ruta de video a iniciar al iniciar el reproductor tras la conexión o URL de youtube a reproducir. Se ha de priorizar la simplicidad y la funcionalidad evitando elementos innecesarios que puedes distraer al usuario.

El diseño de la interfaz se ha pensado siguiendo los principios de usabilidad y mínima interacción, permitiendo que el usuario pueda iniciar la reproducción y sincronización con el rellenado de unos campos mínimos y un click.

Trabajo conjunto.

Como se ha mencionado anteriormente, el desarrollo de dicha aplicación se ha llevado a cabo por dos personas, lo que además del análisis conjunto se necesita una estrategia clara de colaboración y control de versiones para el desarrollo. Para ello, se ha optado por el uso de GitHub, que se trata de una plataforma ampliamente utilizada en el desarrollo de software colaborativo con el fin de gestionar el código fuente, realizar seguimiento de cambios y facilitar la integración continua y ordenada de este.

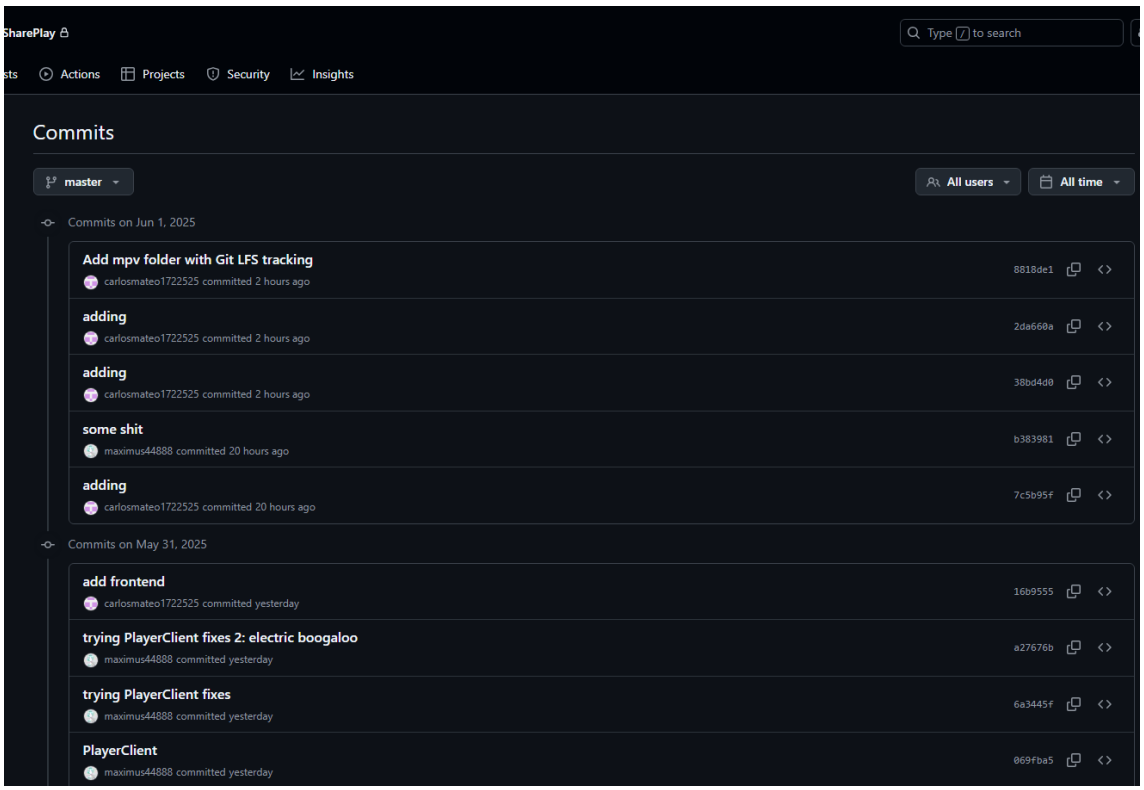
El flujo de trabajo debe de adoptar un flujo de trabajo basado en la creación de un repositorio remoto en la nube, al cual ambos desarrolladores tienen acceso. Cada integrante ha clonado el repositorio en un proyecto local y ha comenzado el trabajo mediante su rama local independiente para posteriormente realizar commits y push sobre el repositorio principal en la nube con el fin de registrar los avances y mantener un historial detallado de los cambios.

Las contribuciones se han integrado al repositorio principal mediante mergeos desde el repositorio local de cada usuario, las cuales han quedado registradas y discutidas en el caso de que hayan tenido conflictos, dejando el código que siga dejando funcional el código desarrollado o la mejor opción aportada antes de fusionarla mediante el merge en la rama principal que nos permite:

- Detectar errores o conflictos antes de que afecten al código principal.
- Hay que asegurar que los cambios estén alineados con los objetivos principales de requisitos de la aplicación de manera ordenada.
- Mantener una trazabilidad clara de quien ha realizado cada modificación y aportación al código.

Además, se han utilizado los comentarios de dichos mergeos como canal de comunicación técnica para ordenar cada aportación y de esa forma tenerlo más organizado.

A continuación, se muestra una imagen del flujo que se ha llevado a cabo en el trabajo de la aplicación en GitHub, donde se puede observar la gestión de ramas, commmits realizados y los pushes.



En esta imagen podemos ver cómo se ha estructurado el desarrollo con un ejemplo de forma ordenada y colaborativa que segura la calidad del código y la coherencia del proyecto y su engrosamiento progresivo.

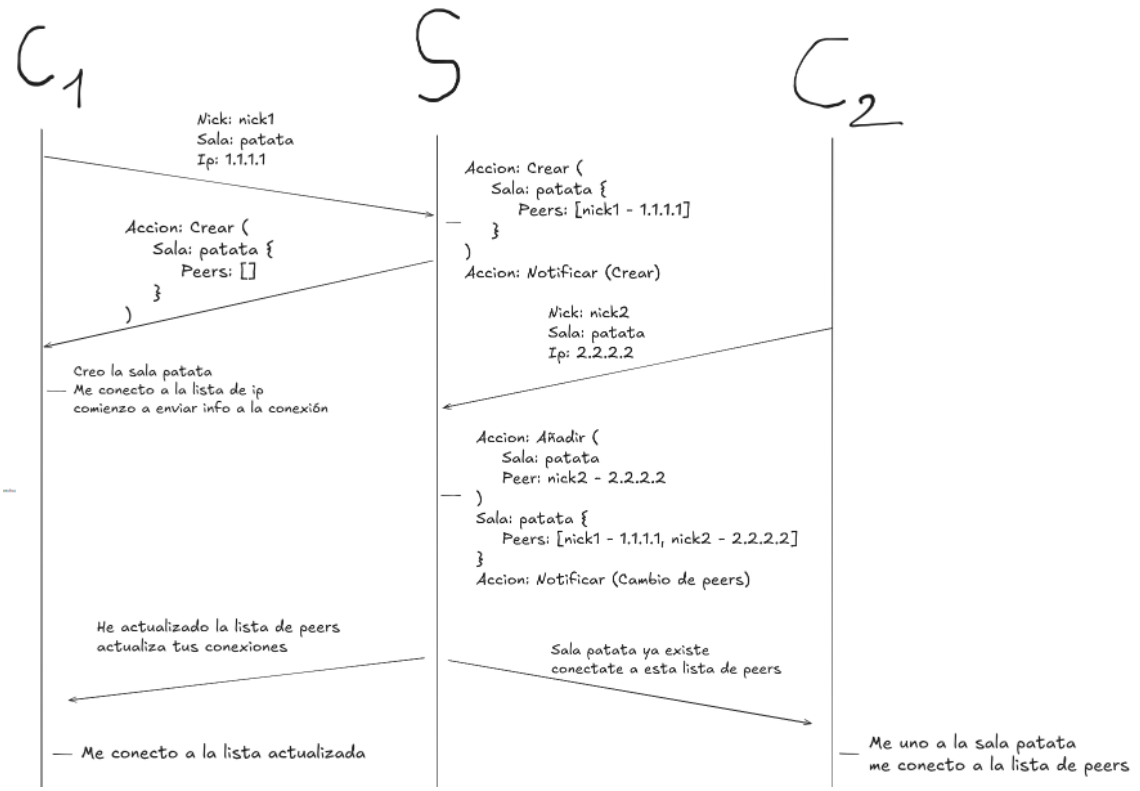
Requerimientos de la aplicación.

Como se ha comentado anteriormente, el objetivo del proyecto tiene un claro propósito desde el principio como se ha descrito: desarrollar una aplicación que permita la reproducción sincronizada de contenido multimedia entre varios usuarios. Sin embargo, era necesario definir como se tenía que llevar a cabo dicho propósito para alcanzar ese objetivo común. Para ello, se realizó llevo un análisis técnico de manera temprana y se consensuó el uso de Kotlin como lenguaje de programación principal de desarrollo.

Kotlin fue elegido por su compatibilidad con el exosistema Java, sus sintaxis moderna, más concisa, facilidad y versatilidad que nos ofrece herramientas avanzadas como las coruotines, que nos permitirá una gestión eficiente y de bajo coste en recursos para la concurrencia. Estas características lo convierten en una opción ideal para el desarrollo tanto de la lógica del cliente como la del servidor, especialmente en aplicaciones que requieren múltiples procesos de manera simultánea y comunicaciones en tiempo real.

Una vez definido el lenguaje, se analizaros los requisitos funcionales y técnicos necesarios para implementar la arquitectura cliente-servidor. El servidor debe de poder desplegarse en la nube en un servidor, en un equipo local con salida a internet y permitir conexiones que en estas opciones sea accesible mediante un dominio o ip, y debe ser capaz de recibir y gestionar múltiples conexiones simultaneas de varios clientes.

A continuación, podemos ver un boceto temprano del esquema general de los componentes y flujos de comunicación entre los diferentes clientes y el servidor.



El servidor permanece en ejecución constante, escuchando por el socket establecido toda conexión entrante por él. Cada cliente que se conecta es gestionado de forma concurrente, lo que significa es que el servidor puede atender múltiples solicitudes al mismo tiempo sin que este bloquee el procesamiento de otras conexiones.

El flujo general debería verse de la siguiente forma:

- El cliente se conecta al servidor.
- El servidor espera recibir el nombre de la sala a la que desea unirse.
- Si la sala no existe el servidor la crea y en caso de que exista la sala indicada añade a dicho cliente a la sala indicada.
- Una vez dentro de la sala, el servidor escucha los eventos enviados por cada cliente.
- Cuando recibe un evento, tiene que mediante lo indicado anteriormente reenviar dicho evento recibido mediante broadcast a todos los demás miembros de esa sala excluyendo al emisor.

Este comportamiento convierte al servidor en un gestor centralizado de salas, conexiones y eventos actuando como intermediario entre todos los clientes, además, de gestionarlos en salas y diferenciando los integrantes de cada sala.

El cliente debe cumplir funciones similares en cuanto a concurrencia:

- Escuchar eventos del servidor en tiempo real.
- Enviar eventos al servidor cuando se detecten cambios en la reproducción.
- Controlar el reproductor MPV, iniciando, pausando o ajustando el tiempo de reproducción.
- Gestionar la interfaz gráfica y las interacciones del usuario.

Esto implica que el cliente debe ejecutar múltiples procesos en paralelo, lo cual también se ha resuelto mediante coroutines para gestionar la concurrencia.

El reproductor utilizado deberá de ser MPV, un reproductor multimedia de código abierto que contiene una potente interfaz de control mediante IPC. Según su documentación oficial (mpv.io/manual/stable), MPV permite iniciar un pipe o un archivo temporal donde se registran eventos del reproductor que le indiquemos y leer los eventos que registra.

El cliente debe:

- Iniciar MPV con un archivo de video local o vacío esperándolo.
- Escuchar continuamente los eventos generados.
- Leer estos eventos desde el pipe cuando se registre dicho evento.
- Enviar os eventos al servidor para que este los propague al resto de la sala.

Este proceso requiere que el cliente tenga al menos varias couroutines dentro del activas, una para leer los eventos del reproductor y otra para comunicarse con el servidor.

Como requisitos de despliegue el cliente deberá de compilarse en un .jar con la interfaz gráfica integrada mediante JavaFX y sus dependencias para su funcionamiento. El servidor debe estar desplegado en un entorno ya sea en la nube o de forma local en un equipo ya que lo permite para permanecer en ejecución y en espera de conexiones o gestionándolas.

Requerimientos de la interfaz.

La interfaz gráfica del cliente representa el punto de interacción directa entre el usuario final y el sistema que hemos estado describiendo. Por ello, se ha diseñado bajo un enfoque en el que destacamos principalmente la simplicidad, claridad y funcionalidad, utilizando JavaFX como tecnología principal para su implementación. JavaFX nos permite construir interfaces modernas, responsivas y fácilmente integrables entre el backend y el frontend con Kotlin, lo que lo convierte en una elección ideal para aplicaciones de escritorio multiplataforma.

La primera vista de la aplicación presenta una ventana no redimensionable con la disposición clara de los elementos necesarios para llevar a cabo una conexión con el servidor que debe incluir:

- Campos obligatorios:
    - Dirección del servidor (host/IP).
    - Nombre de usuario (nickname).
    - Nombre de la sala a la que se desea conectar.
- Campos opcionales:
    - Ruta del archivo de video a reproducir o video de Youtube a reproducir.
    - Campo en el que se puede cargar una configuración mediante código hash.
- Botones funcionales:
    - Iniciar conexión / Ejecutar SharePlay: valida los campos obligatorios y establece la conexión con el servidor, así como guardar configuración actual para el futuro.
    - Limpiar campos: borra todos los campos del formulario y borrado de configuración guardada.

Antes de permitir la conexión, la aplicación debe de llevar a cabo una validación de los campos obligatorio para que estén correctamente rellenado ya que son requeridos para la conexión. En caso contrario, los campos se bordean de rojo con el prompt de requeridos indicando que falta el rellenado de dicho campo.

Si se produce un error en la conexión hacia el host, ya sea porque no existe o porque se ha rechazado dicha conexión se muestra una alerta en el que se especifica que se recomienda verificar el formato y del host o el servidor introducido.

Para mejorar la experiencia del usuario, se deberá de llevar a cabo la implementación de un sistema de persistencia local de la configuración. Como hemos indicado, al pulsar de ejecutar, los datos se guardan y deberán de almacenar en un archivo JSON guardado en los directorios del sistema del usuario que contendrá:

- Dirección de servidor.
- Nombre del usuario.
- Nombre de la sala.

Al iniciar la aplicación, está en caso de que el archivo exista tras su correcta comprobación deberá de rellenar y cargar automáticamente con los valores guardados los campos de conexión. El botón de limpieza permite borrar tanto los campos como el archivo de configuración.

Una vez establecida la conexión con éxito, la aplicación deberá de redirigirnos a la segunda vista donde se visualiza el estado de la sala que incluye:

- Una listview que nos muestra los integrantes de la sala junto con nuestro nombre.
- Un título con el nombre de la sala.
- Un botón volver que cierra la conexión y nos devuelve a la pantalla de conexión.
- Un botón copiar conexión que nos copiara al portapapeles un código de conexión.
- El inicio del reproductor ya sea con el video de YouTube o la ruta del archivo facilitado o vacío en caso de que no se haya pasado.

El reproductor MPV se debe de controlar mediante la interfaz IPC. El cliente escucha los eventos generados y guardados en su respectivo pipe generados de la escucha del reproductor y los transmite al servidor. Para ello, se utiliza el pipe mencionado o archivo temporal que actual como canal de comunicación entre el reproductor y la aplicación.

Este sistema requiere que el cliente gestione múltiples procesos de manera concurrente:

- Escuchar eventos del reproductor.
- Leer los eventos escritos en el pipe.
- Recibir eventos del servidor y aplicarlos al reproductor en consecuencia.
- Gestionar la interfaz gráfica.

Todo esto se implementa mediante coroutines en Kotlin, lo que nos permite una ejecución eficiente y no bloqueante de todas las tareas a tiempo real.

Tecnologías utilizadas y por qué.

A lo largo del desarrollo de esta aplicación y su debido análisis se ha decido usar diversas tecnologías, cada una de estas tecnologías seleccionadas cuidadosamente en función de sus requerimientos para resolver los distintos retos técnicos del proyecto. A continuación, se detallan las principales tecnologías empleadas y las razones de su selección.

Kotlin:

- ¿Por qué? Es un lenguaje moderno, conciso y seguro que corre sobre la JVM. Su compatibilidad con Java y su soporte nativo para programar lo hace en una opción ideal para aplicaciones como esta.
- Ventajas clave: Sintaxis clara, interoperabilidad con Java, gestión deficiente de concurrencia y una comunidad activa.

JavaFX:

- ¿Por qué? Nos permite construir interfaces graficas de usuario modernas y funcionales en aplicaciones de escritorio. Su integración y compatibilidad con Kotlin es fluida y ofrece herramientas visuales potentes para el diseño de interfaces.
- Ventajas clave: Soporte para FXML, CSS, componentes visuales personalizables y multiplataforma.

MPV Player:

- ¿Por qué? MPV es un reproductor multimedia de código abierto que permite control externo mediante comandos y comunicación IPC. Esto lo hace ideal para integrarlo en la aplicación que necesita controlar dicha reproducción desde código.
- Ventajas clave: Ligero, altamente configurable, soporte para scripting mediante código, documentación amplia y extensa.

Sockets TCP/IP:

- ¿Por qué? La comunicación en tiempo real entre cliente servidor requiere de esta tecnología fiable y ampliamente extendida de baja latencia. Los sockets TCP/IP permiten una conexión persistente y bidireccional, ideal para la sincronización de eventos y comunicación.
- Ventajas clave: Comunicación directa, bajo nivel de latencia, control total sobre dicho protocolo de comunicaciones.

Courotines de Kotlin:

- ¿Por qué? La aplicación necesita manejar múltiples tareas concurrentes, como los eventos, nos permiten gestionar estas tareas de forma eficiente y sin bloquear los canales de comunicación.
- Ventajas clave: Bajo consumo en recursos, fácil de implementar y escalabilidad.

Git y GitHub:

- ¿Por qué? Para el trabajo colaborativo entre los desarrolladores, GitHub ha sido la herramienta elegida para el control de versiones, revisión de código, gestión del repositorio y avance del proyecto.
- Ventajas clave: Historial de cambios, trabajo en ramas, mergeos y colaboración remota.

JSON:

- ¿Por qué? Se ha utilizado JSON para almacenar la configuración local del cliente. Es un formato ligero, legible y ampliamente soportado.
- Ventajas clave: Fácil de leer y escribir, integración nativa con el lenguaje de programación usado e ideal para las configuraciones.

IntelliJ IDEA:

- ¿Por qué? Es el entorno de desarrollo más completo para Java y Kotlin, con herramientas avanzadas de depuración, integración con Git y soporte de JavaFX.
- Ventajas clave: Productividad, autocompletado inteligente, integración con herramientas externas.

3. Backend.

Introducción.

Shareplay es un único proyecto en Kotlin divido en los módulos *Common*, *Server* y *Client*. En sendos módulos podemos encontrar código al que podríamos atribuirle la categoría de *backend* y es que, aunque por costumbre el cliente suele ser el *frontend* y el server el *backend*; en nuestro caso y por la naturaleza del proyecto y en parte por ser una aplicación de escritorio, el cliente también tiene mucha lógica de negocio compleja para poder manejar y controlar el reproductor; cosa que no tiene sentido atribuir a *frontend*. Es por esto por lo que no solo hablaremos del servidor en esta sección, sino que también hablaremos del cliente con mucho detalle pues es donde podemos encontrar la lógica mas compleja y clave de Shareplay.

Antes de hablar de las diferentes partes de Shareplay, quisiera dejar claro las dependencias del backend. Están manejadas usando Maven, cada modulo con su propio *pom.xml*, y un *pom.xml* padre en la raíz del proyecto. El *pom.xml* padre define la configuración y dependencias que el resto de los módulos hijos en sus *pom.xml* heredan.

El *pom.xml* padre entonces define las dependencias con la librería estándar de Kotlin, y la librería de corutinas de Kotlin, además de la dependencia con el módulo *Common*. Sendas dependencias son comunes entre los módulos hijos y por eso están en el padre definidos. Además, también define el *build step* necesario para poder compilar correctamente Kotlin de tal manera que los módulos hijos no necesitan definirlo para poder compilar.

Los módulos *Server* y *Client* en sus *pom.xml* también definen un *build step* para poder empaquetarse en *.jar* correctamente.

Servidor.

El servidor es bastante sencillo pues su propósito principal es permitir la intercomunicación entre los distintos clientes separados por habitaciones virtuales; usando corutinas para poder hacerlo concurrentemente. Esto no requiere ninguna dependencia adicional a las ya descritas.

Entrando en los detalles de uso e implementación, al arrancar el servidor este inicia un *ServerSocket* por defecto en *0.0.0.0:1234*. Esta dirección se puede configurar pasando en los argumentos del programa, separado por espacios, la IP y el puerto donde se quiere que el servidor escuche.

Al arrancar correctamente el *ServerSocket*, este entra en un bucle infinito mientras el *ServerSocket* se mantenga abierto, donde se estará esperando la conexión de un cliente. Esta espera no bloquea el thread gracias al uso de corutinas lo cual permite que, aunque el servidor deba atender a otros clientes ya conectados y simultáneamente escuchar por nuevos, no se bloquee el uno al otro. Para esto también se necesita que el manejo de un cliente ya conectado también sea concurrente por lo que nada mas un cliente se conecta, se crea una nueva corutina que de ahí en adelante se encargara de manejar a ese cliente sin bloquear.

En esta corutina, lo primero que se hace es construir un *Room.Client*, Este es un simple objeto que encapsula el *Socket* del cliente, su *ObjectInputStream* y su *ObjectOutputStream*, y su apodo (proporcionado por el propio cliente); además de definir algunos pocos métodos para poder interactuar con estos streams y con ello comunicarse con el cliente.

Este *Room.Client* se pasa como argumento a un método estático suspendible de *Room*, que será la clase que se encargara de manejar las habitaciones virtuales donde los usuarios se pretenden conectar. Este método estático viene acompañado también de un mapa concurrente estático donde se guardan los nombres de las habitaciones como clave, y las instancias de los objetos *Room* como valor. Este método entonces lo que hace es preguntar al cliente *Room.Client* por el nombre de la habitación a la que solicita entrar, comprobar si una habitación con ese nombre ya existe en el mapa concurrente estático (y en caso de no existir crearla), y a través de la llamada a un método suspendible de esta instancia de *Room*, pasarle la instancia de *Room.Client* como parámetro para que empiece a manejar a ese cliente dentro de la habitación que solicita.

Una vez se tiene la instancia de la habitación y se le ha pasado el cliente para que lo maneje, lo primero que hace este método suspendible es añadir dicho cliente a una lista concurrente donde se guardan las referencias a todos los *Room.Client* que esta instancia de *Room* esta manejando actualmente. Seguidamente se notifica a todos los clientes ya conectados de que un nuevo cliente se ha conectado a la sala, especificando el apodo de dicho cliente; y también se notifica al cliente recién conectado de todos los clientes que ya están en la sala previamente; especificando los apodos de todos ellos. De esta manera todos los clientes (nuevos, o que ya estuvieran en la sala) saben los apodos del todos los otros clientes en la sala.

A partir de este punto, este método suspendible entra en un bucle donde lo único que hará es escuchar que dice este cliente, y reenviarlo al resto de clientes. Esta es la parte que permite que, cuando un cliente envie una instrucción de pausa o de salto en el tiempo, le llegue al servidor, y este lo replique al resto de clientes de la sala para que actúen en consecuencia. Estas instrucciones están definidas en el módulo *Common*, el cliente las instancia, las manda al servidor, este las lee y las envía al resto de clientes, los clientes las reciben, y actúan en consecuencia. Para que el servidor pueda leerlas, necesita conocerlas y es por eso que la definición de estas instrucciones se encuentra en el módulo *Common*, bajo la clase serializable *NetworkEvent* que define varias clases que heredan de ella, correspondiendo cada una de ellas con las distintas instrucciones que este pequeño protocolo permite. Cada una de estas instrucciones puedes tener los datos asociados que quiera, permitiendo así pasar diferentes tipos de datos en cada instrucción.

De este bucle se sale cuando el cliente cierre su socket subyacente. Al salir del bucle, el servidor maneja la desconexión del cliente de tal manera que lo borra de la lista concurrente donde se guardan las referencias a todos los *Room.Client* que esta instancia de *Room* esta manejando pues no lo va a manejar más; y comunica al resto de clientes conectados sobre la desconexión del cliente, especificando el apodo de dicho cliente.

Este es el ciclo de vida del servidor, su *ServerSocket*, los distintos *Socket* de los clientes encapsulados en *Room.Client*, y las diferentes instancias de *Room*; y como se puede ver es bastante sencillo en cuanto a la lógica que debe hacer, pero su complejidad emerge del correcto uso de las corutinas para que el servidor sea concurrente; y del correcto manejo de errores que también se complica especialmente trabajando con corutinas.

Cliente.

En el cliente se encuentra mucha más complejidad de la aplicación incluso centrándonos solo en el backend del mismo, pero ante de eso es conveniente resaltar las dependencias del módulo. Especialmente (y obviando las dependencias usadas exclusivamente por la parte del frontend) la dependencia con org.scala-sbt.ipcsocket » ipcsocket. Esta dependencia es clave para el núcleo de funcionalidad del cliente. Se hablará más delante de ello. El cliente también depende de la librería de serialización de Kotlin.

La descripción del propósito del backend en el cliente es mas profunda que la del servidor, pero con ánimo de sintetizar diremos que tiene 2 propósitos:

1. Permitir la comunicación bidireccional y concurrente entre un reproductor multimedia y el propio cliente, permitiendo así:
    a. Que el reproductor comunique al cliente los cambios/acciones/eventos que el usuario de la aplicación esta haciendo sobre el reproductor, y el cliente pueda reaccionar a dichos cambios/acciones/eventos.
    b. Que el cliente pueda comandar o solicitar información al reproductor multimedia, y este pueda ejecutar dichos comandos y dar dicha información de vuelta al cliente.
2. Permitir la comunicación concurrente con el servidor, permitiendo así:
    a. Que al reaccionar el cliente a los distintos cambios/acciones/eventos producidos por el reproductor, esta reacción conlleve enviar los datos necesarios al servidor para que este comunique al resto de clientes conectados a la misma sala los datos necesarios para actuar en consecuencia.
    b. Que, al recibir los datos necesarios del servidor, el cliente pueda actuar en consecuencia y con ello controlar el reproductor multimedia.

Es gracias a estas funcionalidades del cliente (y también con la ayuda del servidor por supuesto), que los distintos clientes de una sala se pueden sincronizar entre si al poder reaccionar a los cambios de sus reproductores locales y comunicar al resto; y al poder reaccionar a los datos del servidor que le comunican el estado del resto de los clientes para poder sincronizarse con ellos.

Toda esta funcionalidad descrita esta programada de tal manera que es agnóstica al reproductor subyacente, que es uno de los grandes propósitos a la hora de desarrollar el cliente. Para conseguir esto, y ya entrando mas en detalles de implementación, se ha desarrollado la clase *PlayerClient*, y la interfaz *Player*.

PlayerClient

PlayerClient es el núcleo/raíz de la implementación. Esta clase se encarga de controlar la comunicación con el servidor bidireccional y concurrentemente; y de reaccionar concurrentemente a los cambios/acciones/eventos del reproductor, además de comandar al mismo que haga ciertas acciones o devuelva cierta información. Todo esto lo consigue usando inteligentemente las corutinas y los métodos y variables expuestas por la interfaz *Player*; además de un manejo de excepciones y de recursos medido con cuidado para obtener el comportamiento esperado.

Para conseguir esto, se debe inyectar en la instanciación de la clase no solo el socket, sino (y más importante) una implementación de la interfaz *Player* que es la se encarga de definir a que cambios/acciones/eventos puede reaccionar el cliente, que comandos puede solicitar el cliente al reproductor hacer, y que información puede solicitar el cliente al reproductor devolver.

De esta manera a *PlayerClient* se le puede inyectar una implementación cualquiera de *Player*, y *PlayerClient* podrá trabajar con ellas sin necesidad de hacer una implementación única de *PlayerClient* por cada reproductor con el que queramos que el cliente sea compatible.

PlayerClient también se encarga de comunicar al servidor el apodo del usuario, y la habitación a la que quiere unirse, además de exponer la lista de clientes conectados a la sala y un método para poder cargar un fichero (estas dos últimas cosas son para el uso del frontend para poder mostrar en la vista los usuarios actuales en la sala, y para permitir cargar un video inicial al arrancar el reproductor)

El cliente entonces necesita definir implementaciones de *Player* y, en este caso hay una implementación para el reproductor MPV. Esta implementación es sin duda alguna donde se encuentra la mayor complejidad de toda la base de código de Shareplay, y donde mas tiempo se ha invertido en el desarrollo de la aplicación.

MPV.

La clase *MPV* implementa la interfaz *Player* para permitir la comunicación y control del reproductor multimedia MPV. Esto requiere del uso de ciertas dependencias.

Para poder usar MPV para el propósito de la aplicación, necesitamos saber que maneras proporciona este reproductor para poder comunicarnos/controlarlo desde otro proceso. Y en el caso de MPV nos permite usar JSON IPC. El como se usa y cómo funciona este protocolo esta descrito en detalle a lo largo de toda la documentación de MPV y no es pertinente explicarlo aquí más allá de lo necesario para entender la implementación.

En esta parte es donde entra la dependencia con org.scala-sbt.ipcsocket » ipcsocket y es que MPV JSON IPC en Windows usa *named pipes*. Son unos ficheros especiales en los que distintos procesos pueden leer y escribir bidireccionalmente full-duplex. En este fichero, MPV escribe (en formato *JSON*) los cambios/acciones/eventos que ocurren en el reproductor y lee (también en formato *JSON*) los comandos que el cliente escribe en ese mismo fichero que a su vez también es capaz de leer de ese mismo fichero para saber que ocurre.

Como esta comunicación a través de este *named pipe* tiene que ser bidireccional y full-duplex, el cliente necesita una abstracción correcta que permita esto sin bloquearse a si mismo. Este fue uno de los grandes problemas durante el desarrollo de la aplicación y es que resulta que ninguna de las clases de la librería estándar de Java ni de Kotlin funcionan correctamente con las *named pipe* de Windows. Es por esto por lo que se necesita la dependencia con org.scala-sbt.ipcsocket » ipcsocket que proporciona una implementación de *Socket* para poder trabajar con este tipo de ficheros y en general para usar *IPC* (*Inter-Process Communication*).

Los detalles de implementación de esta clase son bastante complejos. Para empezar (y recordando el uso constante de concurrencia con corutinas, y el correcto manejo de errores y recursos), al instanciar la clase se arranca el proceso de MPV con los parámetros necesarios para arrancar el servidor *IPC* (y para poder usar *yt-dlp*, cosa que se explicará más adelante). Una vez arrancado el proceso e iniciado el servicio IPC del reproductor, se crea el *Socket* con la implementación de org.scala-sbt.ipcsocket » ipcsocket conectándose al *named pipe*, y su *Reader* y *Writer*.

A nivel superficial, esta clase tiene un bucle que lee continuamente todo lo que se escriba en el *named pipe* (concurrentemente para no bloquear) usando *SharedFlow*s de Kotlin. Este lee del *Reader*, lo parsea a un objeto JSON de la librería estándar de serialización de Kotlin, y en función del contenido completa ciertas promesas pendientes (que se producen cuando es el cliente el que hace una petición de datos o el que ha comandado una acción), o reemite objeto JSON al *SharedFlow*.

Los *SharedFlow*s actúan como productores con una cola de subscripciones, es decir, al instanciarlo se dice que tipo de dato produce, y diferentes subscriptores pueden suscribirse para obtener dicho dato. Con esta arquitectura, estará este *SharedFlow* que se encarga de producir objetos JSON directamente del *named pipe*, y luego las diferentes partes de la clase que necesiten los datos que se están produciendo en el *named pipe* se suscribirán a este *SharedFlow* y tendrán acceso a esos datos. En vez de intentar describir un caso, adjunto una imagen con seudocódigo donde se puede entender la arquitectura. (Resaltar que es seudocódigo y que no representa todo lo que el código real hace. Es una simplificación)

```
incoming = flow {
    while true {
        incomingMessage = reader.read()
        parsedMessage = JSON.parse(incomingMessage)
        // Emite el valor en este flow productor
        // para todos los consumidores suscritos
        emit(parsedMessage)
    }
}

pauseEvents =
    incoming
    .filter(parsedMessage -> isPauseEvent(parsedMessage))
    .map(pauseMessage -> mapToPauseState(pauseMessage))

seekEvents =
    incoming
    .filter(parsedMessage -> isSeekEvent(parsedMessage))
    .map(seekMessage -> mapToSeekTime(seekMessage))
```

La variable *incoming* es el productor de objetos JSON producidos por la lectura directa del Reader del *named pipe*. Y luego otros *SharedPipe*s (como son *seekEvents* y *pauseEvents*) se pueden subscribir a él y de esta manera, cuando *incoming* produzca un valor (porque ha leído uno del *named pipe*, porque el MPV ha escrito en dicho *named pipe*), estos subscriptores/consumidores serán notificados, comprobaran si ese objeto JSON es el que están esperando para producir ellos sus propios valores, y de serlo mapearan y producirán el valor que necesiten (en el caso de los eventos de pausa, un booleano representando el estado de pausa; y en el caso de los eventos de salto, un objeto *Duration* de Kotlin representando el instante de tiempo al que se ha saltado). Con esta arquitectura se puede ver cómo es fácil expandir los eventos que esta clase escucha sin necesidad de modificar ni entremezclarse con el código de otros eventos.

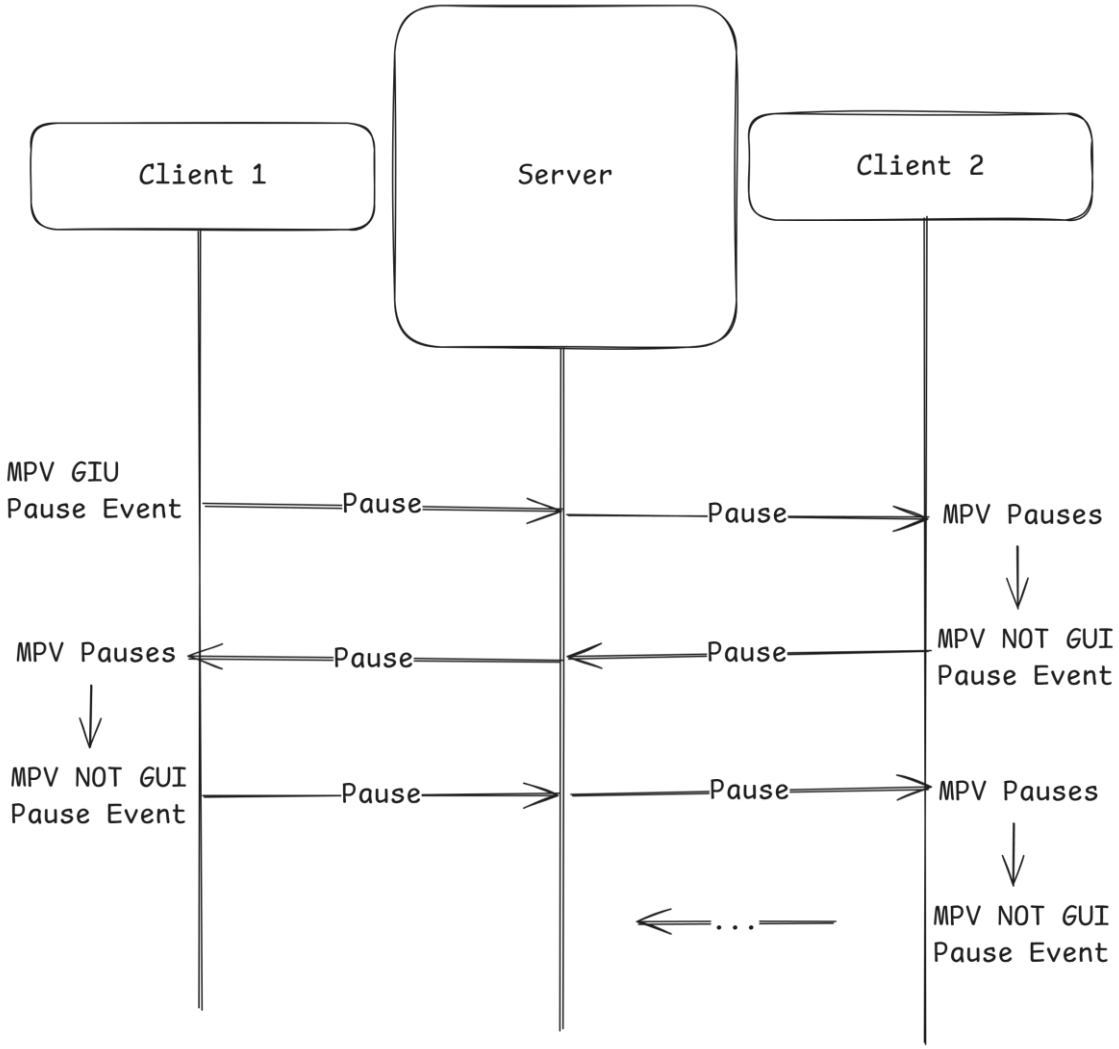
La clase *MPV* también implementa los métodos necesarios por le interfaz *Player* para poder cambiar el fichero en reproducción, y pausar y resumir el reproductor. Esto requiere de un sistema de escritura en el *named pipe* y es que, toda escritura en él, vendrá correspondida con un mensaje de respuesta informado sobre la correcta ejecución del mismo por parte de MPV.

Con el objetivo de que, cuando se hagan estas peticiones a MPV el cliente consciente sobre la correcta ejecución del mismo y espere a la respuesta, pero sin bloquear el hilo; se hace uso de *CompletableDeferred*. Se adjunta seudocódigo para visualizar.

```
pending: Map<Int, CompletableFuture>
incoming = flow {
    while true {
        incomingMessage = reader.read()
        parsedMessage = JSON.parse(incomingMessage)
        if isResponseToRequest(parsedMessage) {
            // Completes the deferred value for the pending request
            pending[parsedMessage.id].complete(parsedMessage)
        }
    }
}
execute(request) {
    deferred = CompletableDeferred()
    pending[request.id] = deferred
    writer.write(request)
    // Wait until incoming completes the deferred value
    return deferred.await()
}
pause() {
    pauseRequest = buildPauseRequest()
    execute(pauseRequest)
}
seek(time) {
    seekRequest = buildSeekRequest(time)
    execute(seekRequest)
}
```

Con el seudocódigo se ve mejor como, al enviar una request al MPV, esto no devuelve hasta que sea completado por el productor global *incoming*. De hecho, es precisamente por este productor global que se debe encargar exclusivamente de la lectura del *named pipe* que no puede cada uno de los métodos (*pause* y *event*) esperar “manualmente” al mensaje de respuesta; y por lo que se deben usar *CompletableDeferred*.

Por último, quisiera resaltar otra peculiaridad de la implementación de *MPV*, y es que el cliente al comandar a MPV a pausar/resumir/saltar; esto produce un evento de pausa/salto que los *SharedFlow*s descritos anteriormente consumirán, pero realmente el comportamiento esperado es que estos *SharedFlow*s solo produzcan sus eventos correspondientes si y solo si han sido producidos por el propio usuario al interactuar con la GUI de MPV. De lo contrario habría un problema en el uso de la aplicación tal y como se puede ver en el siguiente diagrama.



El diagrama es un ejemplo del comportamiento de Shareplay cuando ocurre una pausa (que también ocurre igual con los saltos y con el *resume*) si no se hace un sistema para evitarlo. De no hacerlo, se enviarán paquetes "infinitamente". Es más, la cantidad de paquetes escala exponencialmente con el número de clientes en la sala. Y aunque hay varias maneras de solucionarlo, la más correcta es hacer que no se produzcan eventos cuando estos son resultado de comandos del cliente, y no te interacciones del usuario con la interfaz.

Describir los detalles de implementación de la solución a este problema es complicado, pero conceptualmente la solución consiste en tener contadores de cuantos "comandos del cliente" se han hecho, y luego cada *SharedFlow*, cuando se produce un evento correspondiente a lo que espera, entonces comprueba si dicho contador es mayor de 0, de serlo, lo decrementa en uno, y dropea el evento. Con esto siempre se dropea un evento producido por MPV justo después de que el cliente haya comandado a MPV una acción; evitando el problema del diagrama.

3. Frontend.

Introducción.

La interfaz de usuario representa uno de los elementos clave de la interacción del usuario final con la aplicación. En este proyecto, se ha optado por un enfoque de diseño minimalista y funcional, priorizando la simplicidad de suso sin sacrificar la potencia de las funcionalidades que ya son ofrecidas. Por ello, se ha utilizado JavaFX como una tecnología principal para el desarrollo del front-end, junto con la combinación del patrón Modelo-Vista-Controlador que esta tecnología implementa, lo que nos permite una separación clara entre la lógica de presentación, la lógica de negocio y los datos tratados.

La interfaz se compone de dos vistas principales:

- Vista de configuración inicial donde los usuarios introducen los parámetros necesarios para estableces la conexión con el servidor y la configuración de sesión.
- Vista de sala donde se gestiona la interacción con el resto de los usuarios, el estado de dichas salas y el control del reproductor MPV.

Ambas vistas están definidas con los archivos de control normalizados por JavaFX que son los ficheros FXML, lo que nos permite una estructuración declarativa de dicha interfaz. Cada vista estará asociada a su respectivo controlador que lo gestiona, encargado de gestionar los eventos de la interfaz, validar los datos introducidos y coordinar la lógica de conexión y reproducción.

Primera vista (pantalla de inicio):

- Campos de texto para:
    - Dirección del servidor.
    - Nombre de usuario.
    - Nombre de la sala.
- Campos opcionales:
    - Ruta del archivo de video o URL de YouTube.
    - Campo de carga de otra configuración hecha de conexión.
- Botones:
    - Ejecutar SharePlay y guardar configuración actual.
    - Limpiar campos.

Segunda vista (pantalla de sala):

- ListView con los usuarios conectados.
- Reproductor MPV embebido o controlado externamente.
- Botón para volver a la vista inicial, cerrando la conexión y el reproductor.
- Botón para copiar configuración actual establecida.

Para mejorar la estética y la experiencia de usuarios, se ha aplicado en la aplicación una hoja de estilos CSS común entre ambas vistas, esta se guarda y ubica en los resources del proyecto. Esta hoja define estilos modernos y coherentes para botones, campos de texto, listas y otros componentes visuales, proporcionando una apariencia limpia y profesional.

Los datos introducidos por el usuario son guardados y almacenados localmente en un archivo JSON, ubicado en el directorio del sistema del usuario. Esta funcionalidad nos permite que, al reiniciar la aplicación, los campos se rellenen automáticamente con el inicio de la aplicación y dicha configuración guardada previamente, esto mejora la usabilidad y la reducción la fricción entre el uso repetitivo de la aplicación permitiendo conexiones más rápidas.

El flujo entre vistas está gestionado desde los controladores que se deben de encargar de:

- Validar los campos obligatorios antes de permitir la conexión.
- Mostrar alertas informativas en caso de errores (La conexión al servidor o formato de este).
- Cargar dinámicamente la segunda vista al establecerse la conexión.
- Cerrar la conexión y el reproductor al volver a la vista inicial.
- Copiar configuración hecha de la conexión para compartir dicha conexión mediante un código.

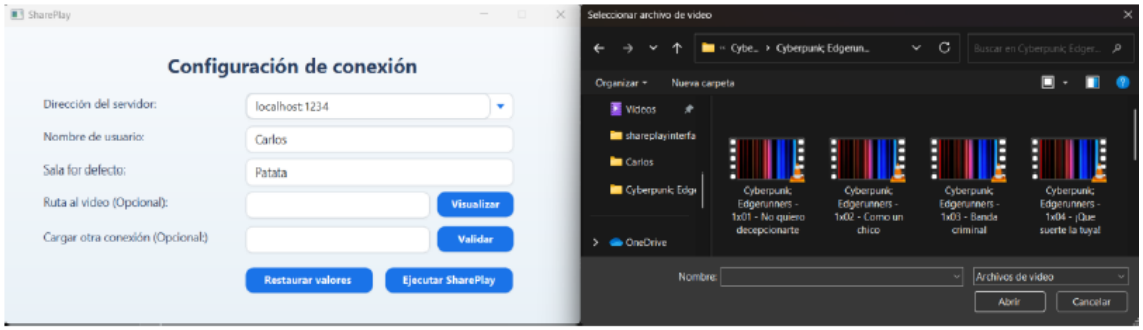
Vistas.

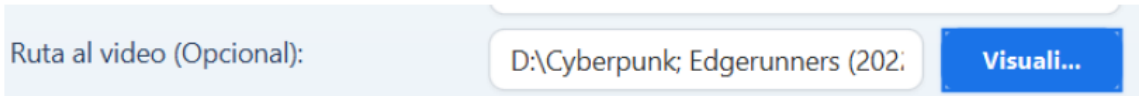
La interfaz gráfica de la aplicación se estructura en dos vistas principales, estas han sido desarrolladas mediante ficheros FXML y gestionadas por sus respectivos controladores en su gestión. Dichas vistas representan los dos estados fundamentales de la aplicación: la configuración inicial de la conexión y la gestión de la sala reproducción compartida.

Esta vista es la primera pantalla que el usuario encuentra al iniciar dicha aplicación. Su objetivo es permitir al usuario introducir los datos necesarios para que se establezca dicha conexión con el servidor y configuración de la sesión de reproducción.

En esta interfaz se presentan los siguientes elementos descritos:

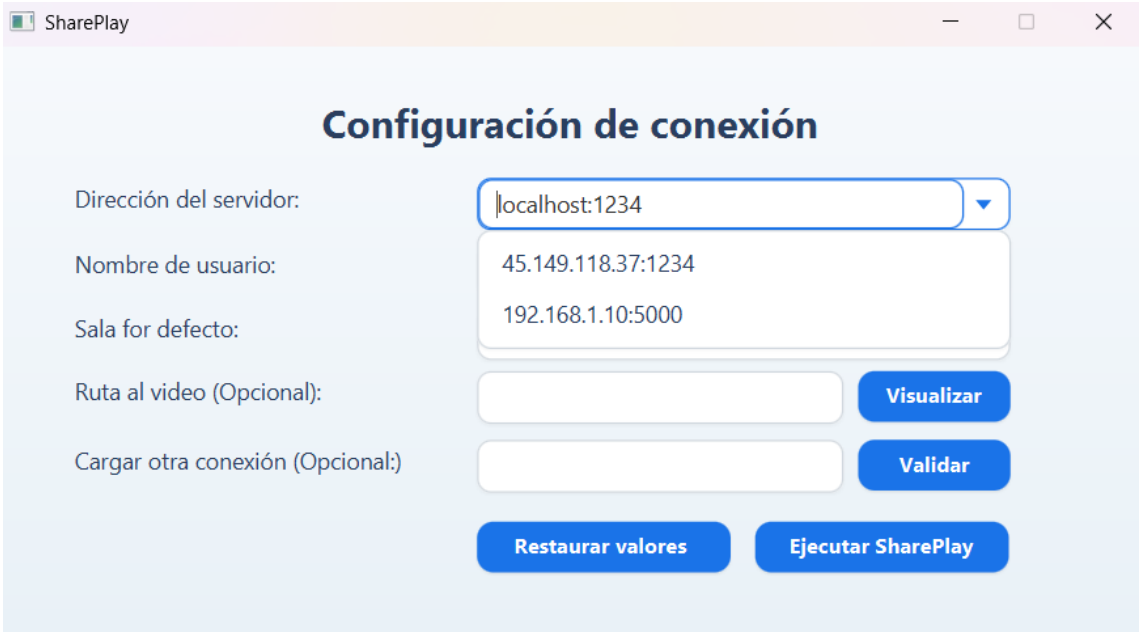
- Campos obligatorios:
    - ComboBox para seleccionar o introducir la dirección del servidor.
    - TextField para el nombre de usuario.
    - TextField para el nombre de la sala.
- Campos opcionales:
    - Ruta del archivo de video a reproducir o URL de video de Youtube.
    - Código para validar configuración y cargarla.
- Botones funcionales:
    - Visualizar: abre un explorador de archivos para seleccionar un video.
    - Ejecutar SharePlay: valida los campos y establece la conexión y guarda la configuración en caso de que se establezca.
    - Restaurar valores: borra todos los campos del formulario.
    - Validar: valida el código y carga los campos de la conexión.





Como vemos se nos permite visualizar la ruta al archivo y seleccionar un fichero de reproducción para en el caso de que se desee se inicie el reproductor con dicho fichero ya.

Además, se incluye un campo adicional para introducir un código de conexión compartido. Este código generado por otro usuario contiene la información codificada que nos permite descifrar y cargar el servidor y la sala en nuestra configuración. Al pulsar el botón validar el sistema lo descodificara y los fields se rellenarán automáticamente los campos correspondientes.



El comboBox permite seleccionar entre servidores predefinidos o introducir también manualmente la dirección del servidor que hayamos desplegados nosotros mismos.

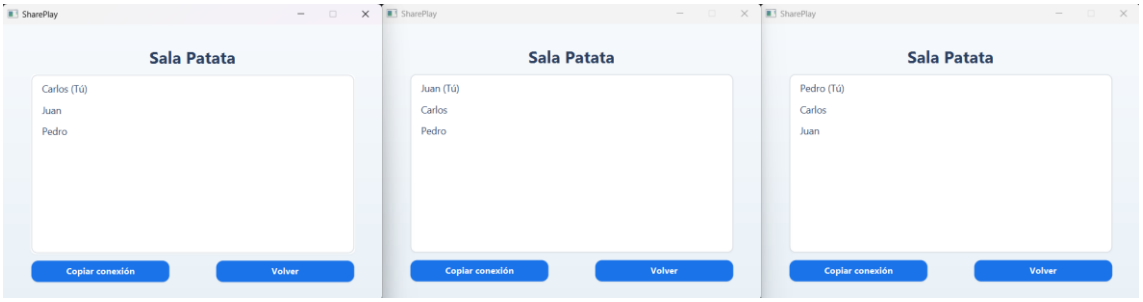
Una vez completados los campos obligatorios, si el servidor está disponible y responde correctamente a esta conexión se establece la conexión y comienza con la carga de la segunda vista e inicio del reproductor. En caso de error, se mostrará una alerta informativa indicando que no ha podido conectar con el servidor o que el formato de este es erróneo.

Este mecanismo de validación de código nos facilita la incorporación rápida de nuevos usuarios a una sala ya existente, sin necesidad de introducir manualmente los datos.

Una vez establecida la conexión, se accede a la segunda vista, donde se gestiona la sala, integrantes y la reproducción compartida y sincronizada con el resto de los integrantes. En esta pantalla se inicia el reproductor MPV, que puede comenzar con el video seleccionado previamente o bien permaneces en espera de que se cargue un video que le demos al reproductor.

En esta vista como podemos ver se muestran:

- Una ListView con los usuarios conectados incluyéndonos a nosotros mismos de la sala actual.
- Un título que incluye el nombre de la sala a la que se ha conectado dentro del servidor.
- Un botón para volver a la vista inicial, que cierra la conexión con el servidor y detiene el reproductor MPV.
- Un botón que copia la configuración actual de conexión para poder facilitárselo a otra persona y que haga la conexión, facilitando así la sincronización entre diferentes clientes.



Cada usuario ve en su equipo local el estado actualizado de los en el que se ven los demás integrantes de la sala viendo y actualizándose en todo momento los integrantes de esta para saber quién hay en la sala, permitiendo una experiencia de visualización compartida y sincronizada.

Controladores.

Los controladores son los componentes encargados de la gestión de la lógica de interacción entre la interfaz gráfica definida en ficheros FXML y el backend de la aplicación. Cada vista cuenta con su propio controlador, que implementa los métodos necesarios para responder a las acciones del usuario, validar dichos datos, gestionar la conexión con el servidor y controlar el reproductor MPV.

Este controlador gestiona todos los elementos de la pantalla de configuración. Cada botón de la interfaz está vinculado con un método especifico que al momento de ejecutarlo la lógica correspondiente de gestión de datos.

Las principales funcionalidades implementadas en este controlador son la siguientes:

- Carga de configuración previa: al iniciar la aplicación, se debe comprobar si existe un archivo de configuración JSON en los archivos del usuario del sistema. Si es así, se cargan automáticamente los valores en los campos de host, nombre de usuario y sala dejando preparado la conexión.
- Validación de campos: al pulsar el botón Ejecutar SharePlay, se valida que los tres campos obligatorios estén correctamente rellenados. Si alguno está vacío, se muestra una alerta de error mediante cambiar el CSS de los bordes de los campos y rellenándolos con un prompt de requerimiento.

- Gestión de errores de conexión: si el campo de host contiene una cadena no vacía mediante una validación Empty, pero no válida esta se muestra una alerta indicando que no se ha podido establecer la conexión coloreando los campos de rojo en sus bordes y poniendo un promopt de requerimiento.
- Transición a la segunda vista: si la conexión es exitosa, se pasa el socket de conexión, el path del archivo de video si se lo hemos pasado o el de youtube si le hemos pasado una URL a un video, el nickname y el nombre de la sala al controlador de la segunda vista. La vista actual se cierra y se carga la nueva.
- Explorador de archivos: el botón visualizar abre un explorador de archivos para seleccionar un video. Una vez seleccionado, se actualiza el campo correspondiente con la ruta del archivo y se comprueba que se haya seleccionado un fichero que de video.
- Validación de código compartido: el botón Validar descodifica un código introducido que nos ha facilitado otro usuario que ya está conectado, rellenando automáticamente los campos de host y sala. Esto facilita la incorporación rápida a una sala compartida.

Una vez hecha la gestión de configuración local que lo que hará será guardar la configuración de conexión actual en el archivo JSON que se ha creado en el sistema del usuario y restableces la configuración en el que se elimina el JSON y limpia los campos del formulario.

Este controlador se activa una vez establecida la conexión con el servidor. Su función principal es la de gestión de la interacción del usuario dentro de la sala y controlar el reproductor MPV así como la de escuchar sus cambio.

Las funcionalidades clave de este controlador son la siguientes:

- Inicialización de la conexión: al recibir el socket y los parámetros desde la vista anterior, se inicia la conexión con el servidor, se identifica al cliente y se une a la sala correspondiente imprimiéndolo en su vista, así como en la de los demás integrantes en su vista local.
- Inicio del reproductor MPV: si se ha proporcionado un archivo de video o una URL de un video de Youtube, se inicia la reproducción automáticamente. Si no, el reproductor queda en espera de que se cargue un video.
- Gestión de la lista de usuarios: se recupera un array con los integrantes de la sala. Este array se convierte en un ObservableList, que se vincula a la ListView para mostrar en tiempo real los usuarios conectados y su estado incluyéndonos a nosotros mismo e indicando que somos nosotros.
- Un título en el que se visualiza el nombre de la sala a la que hemos establecido la conexión, al poder crear salas con cualquier nombre, este título se centra.

- Control del reproductor: el reproductor MPV se ejecuta desde una ruta local. Si no existe, se extrae desde el ejecutable del cliente y se guarda para futuras ejecuciones. Desde este punto, el controlador gestiona la sincronización de eventos como reproducción, pausa y saltos de tiempo dentro del código en el que se gestiona.

- Botón Volver: al pulsarlo, se cierra la conexión con el servidor, se elimina al cliente de la sala y se destruye la vista actual, volviendo a la pantalla de configuración inicial de la que hemos hablado anteriormente.

- Botón Copiar configuración: genera un código codificado con los datos actuales de conexión y lo copia al portapapeles. Este código puede ser compartido con otros usuarios para facilitar su incorporación a la misma sala para que posteriormente sea descodificado y carguen la configuración de la sala de una manera más rápida para facilitar la conexión.

Persistencia y control de JSON.

Una parte fundamental de la experiencia para con el usuario en esta aplicación que hemos desarrollado es la capacidad de recordar configuraciones previas para facilitar futuras conexiones. Para ello, se ha implementado un sistema de persistencia local, en los archivos del sistema del usuario, basado en archivos JSON, que nos permite guardar y recuperar automáticamente dos datos introducidos por el usuario para establecer la conexión.

Cada vez que se establece una conexión exitosa con el servidor, la aplicación guarda los siguientes datos:

- Dirección del servidor (host).
- Nombre de usuario.
- Nombre de la sala.
- Ruta del archivo de video (si se ha proporcionado).
- Código para descodificar para cargar otra configuración (si se ha proporcionado).

Estos datos se encapsulan en una clase de datos propia de Kotlin, y se serializan en un archivo llamado .configuracion.json, ubicado en el directorio del sistema de dicho usuario. Este archivo actúa como una memoria persistente de la última sesión establecida con un servidor.

Al iniciar la aplicación, el controlador de la vista inicial comprueba la existencia del archivo .configuracion.json. Si está presente entonces, se deserializa su contenido para que sea cargado en el formulario de dicha vista principal de manera automática. Esto permite que el usuario pueda reconectarse rápidamente sin necesidad de volver a introducir los datos manualmente.

- Guardado: se realiza automáticamente cada vez que se establece una conexión. Se utiliza una clase utilitaria que gestiona la serialización de los datos y la escritura en el archivo JSON.
- Borrado: al pulsar el botón de restablecer configuración, se elimina el archivo .configuracion.json y se limpian todos los campos del formulario, restaurando el estado inicial de la aplicación.

Un mecanismo similar se aplica al reproductor MPV. La primera vez que se inicia la reproducción, los archivos necesarios para ejecutar MPV se extrae del cliente y se almacenan en una ruta fija dentro de los archivos del sistema del usuario, los archivos necesarios para ejecutar el reproductor MPV se extraen del ejecutable del cliente y se almacenan en la ruta fija indicada. Esta ruta se reutiliza en futuras ejecuciones, evitando la necesidad de crear archivos temporales cada vez que se lanza el reproductor.

Este enfoque mejora el rendimiento y reduce la redundancia de datos, asegurando que el reproductor este siempre disponible sin necesidad de reinstalación.

Para facilitar la manipulación de los archivos JSON, se ha utilizado una librería de serialización que nos permite mapear directamente los objetos Kotlin a estructuras JSON y viceversa. Esto elimina la necesidad de acceder manualmente a claves o recorrer estructuras anidadas simplificando el código y reduciendo la posibilidad de errores.

Errores, problemas encontrados.

Durante el desarrollo de este front-end de la aplicación se identificaron y nos dimos cuenta durante el desarrollo de diversos desafíos técnicos, problemas y de diseño que requirieron soluciones específicas para garantizar una experiencia de usuario fluida y un funcionamiento correcto del sistema descrito. A continuación, se detallan los principales problemas encontrados y las decisiones adoptadas para resolverlos.

Uno de los primeros problemas que nos percatamos fue de la necesidad de controlar un único tipo de reproductor, en este caso el MPV desde la aplicación. Esto implicaba que el sistema debía conocer la ruta exacta de ejecutable MPV para poder inicializarlo y controlarlo mediante el sistema de comandos que se ha creado.

- Problema: permitir al usuario introducir manualmente la ruta del reproductor podía generar errores si se proporcionaba una ruta incorrecta o apuntaba a un reproductor no compatible dado que la aplicación está pensada y gestiona MPV de código abierto.
- Solución: se decidió incluir el ejecutable de MPV dentro del propio paquete de la aplicación. Al iniciar la reproducción por primera vez, el programa extrae los archivos necesarios y los guarda en una ruta fija dentro del sistema del usuario. En ejecuciones posteriores, se reutiliza esta ruta, evitando la creación de archivos temporales redundantes.

Esta solución nos garantiza que siempre se utiliza una versión compatible del reproductor y evita errores derivados de esta misma y de las configuraciones externas.

Otro desafío que se nos presenta fue la necesidad de mantener cierta persistencia de datos, si bien no la necesitamos para los usuarios dado que no tienen cuenta, es un ejecutable local y por tanto no necesitamos una BBDD, sí que necesitamos cierta persistencia para la lógica de la aplicación.

- Problema: sin persistencia, el usuario tendría que introducir manualmente los mismos datos en cada ejecución, algo que puede llegar a cansar.
- Solución: se implementó un sistema de persistencia local mediante archivos JSON, que almacena la última configuración utilizada que se ha establecido con un servidor. Este archivo se guarda en el directorio del usuario del sistema y se carga automáticamente al iniciar la aplicación. Si el usuario desea restablecer la configuración, puede eliminar este archivo mediante la funcionalidad desde la propia interfaz.

La validación de los campos de entrada fue otro reto para evitar errores de conexión y mejorar la experiencia de usuario con la aplicación.

- Problema: si los campos obligatorios (host, nombre de usuario, sala) no estaban correctamente rellenados, la aplicación podía fallar o comportarse de forma inesperada debido a la falta de información para llevar a cabo la conexión con el servidor.
- Solución: se implementó una validación estricta. Los tres campos obligatorios deben estar completos para habilitar la conexión. El campo del host debe contener una dirección válida y accesible. Para ello, se realiza una prueba de conexión al socket antes de continuar. Si la conexión falla, se muestra una alerta informativa al usuario indicándole el motivo de la falla.

Inicialmente, cada vez que se iniciaba una reproducción, se generaba un archivo temporal para ejecutar MPV, lo que podía provocar la acumulación en los archivos del sistema del usuario un eventual llenado del disco.

- Problema: generación excesiva de archivos temporales en cada ejecución, haciendo redundante los archivos en el sistema del usuario.
- Solución: se modificó la lógica para que el reproductor se extraiga una única vez y se almacene en una ruta fija. En futuras ejecuciones, se reutiliza esta ruta, evitando la creación de archivos adicionales, comprobando en el momento de la conexión que dicha ruta exista, si no, ejecuta la acción de extraer el ejecutable del reproductor.

El diseño de la aplicación implicaba con ello que un único objeto o cliente gestionando tanto la conexión con el servidor, así como el control del reproductor.

- Problema: era necesario pasarle al objeto la ruta del reproductor, pero permitir que el usuario la introdujera manualmente podía generar errores.
- Solución: se eliminó la necesidad de que el usuario proporcione esta ruta, integrando el reproductor el buscar el fichero a reproducir para que tenga que seleccionarlo y la ruta sea la absoluta y la correcta.

4. Conclusión.

El desarrollo de esta aplicación ha dado como resultado esperado una herramienta totalmente funcional y eficaz que nos permite a varios usuarios reproducir contenido multimedia de forma sincronizada desde sus respectivos equipos de forma local y a distancia en diferentes redes. La solución propuesta responde a una necesidad concreta que es la de compartir experiencias de video a distancia, manteniendo la sincronización en todo momento entre los participantes sin necesidad de compartir archivos ni depender de plataformas externas para ello.

La aplicación destaca por su sencillez y minimalismo en la interfaz para su uso, una interfaz intuitiva y su arquitectura robusta basada en las tecnologías modernas como son las de Kotlin, JavaFX, MPV, Sockets TCP/IP y coroutines. Además, se ha implementado un sistema de persistencia local mediante la aplicación y desarrollo de archivos JSON, lo que mejora la experiencia final del usuario al recordar configuraciones ya hechas y la facilidad de las siguientes repetitivas sin necesidad de una persistencia en una BBDD.

Durante el desarrollo de dicha aplicación se han abordado diversos y variados retos técnico, como la gestión del reproductor MPV, la validación de campos de formulario, la persistencia de datos y la optimización del uso de recursos del sistema, así como su persistencia. Cada uno de estos desafíos ha sido resuelto con soluciones eficientes y escalables, lo que ha permitido la construcción una base sólida y eficiente para futuras versiones.

Aunque la aplicación cumple plenamente con los requisitos funcionales definidos se ha identificado durante el desarrollo de la esta varias áreas de mejora que podrían implementarse en versiones posteriores.

- La compatibilidad con múltiples reproductores: permitir que el usuario utilice otros reproductores además del MPV, manteniendo el control y la sincronización tal y como lo hace actualmente.
- Sincronización automática de archivos: implementar un sistema que permita a los nuevos usuarios buscar automáticamente el archivo de video en su sistema local, eso debería de basarse en el nombre del archivo que están reproduciendo los demás usuarios de la sala.
- Sistema de chat integrado: Deberá de añadir un canal de comunicaciones textual extra dentro de la sala para facilitar la coordinación entre los usuarios sin depender de herramientas externas que son las que soportan la actual carga de interacción con los usuarios antes de conectarse.
- Distribución multiplataforma: Se ha pensado en generar ejecutables nativos para los sistemas operativos como Windows y Linux, eliminando la necesidad de tener instalada la máquina virtual de Java para que funcione la aplicación.
- Mejoras en la interfaz: Seguir refinando la experiencia de usuario con animaciones, notificaciones y una interfaz más dinámica para la mejora final con los usuarios.

Este proyecto no solo ha cumplido con su objetivo final y funcional, sino que también ha sido diseñado con una arquitectura modular y escalable, lo que nos facilitara su mantenimiento y evolución a lo largo del tiempo. Tanto el cliente como el servidor han sido desarrollados como una estructura clara y flexible, permitiendo su despliegue en entornos locales o en la nube con configuraciones mínimas y sencillas.

El uso de GitHub como plataforma de control de versiones nos garantiza y ha permitido un desarrollo colaborativo entre todas las partes implicadas de manera ordenada, con un seguimiento detallado de los cambios y una integración continua que ha favorecido la calidad del código y su fácil continuidad.

En definitiva, esta aplicación representa una solución innovadora y accesible para compartir experiencias audiovisuales a distancia. Su diseño modular, su enfoque en la usabilidad de usuario y su potencial de crecimiento la convierten en una base muy sólida para las futuras versiones más completas y con una futura mejor versatilidad. Este trabajo demuestra cómo, con una planificación adecuada, el uso de tecnologías modernas y una visión, clara es posible desarrollar soluciones funcionales, escalables y con un impacto real en la forma en la que las personas se conectan y comparten momentos sin necesidad de esta juntos.