



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería
Informática**

**Sistema de Procesamiento de
Vídeo**



Presentado por Álvaro Márquez
en Universidad de Burgos — 7 de julio de 2025
Tutores: D. José Miguel Ramírez Sanz y D.
José Luís Garrido Labrador



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



D. José Miguel Ramírez Sanz y D. José Luís Garrido Labrador, profesores del Departamento de Ingeniería Informática, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Álvaro Márquez, con DNI 71362377R, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado Sistema de Procesamiento de Vídeo.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 7 de julio de 2025

Vº. Bº. del Tutor:

Vº. Bº. del co-tutor:

D. José Luís Garrido Labrador

D. José Miguel Ramírez Sanz

Resumen

En el contexto tecnológico actual, la generación de datos de vídeo ha crecido de forma exponencial, creando la necesidad de desarrollar sistemas robustos, escalables y eficientes para su procesamiento. Este proyecto aborda dicho desafío mediante el diseño y la implementación de una infraestructura de software completa y de extremo a extremo, desde la captura del vídeo hasta su análisis.

El sistema utiliza Jitsi Meet como plataforma de videoconferencia y su componente Jibri para la grabación de las sesiones, generando los ficheros de vídeo fuente. Para la ingesta y el procesamiento, se ha desplegado un pipeline de datos que utiliza Apache Kafka como broker de mensajería en su modo moderno KRaft y un clúster de Apache Spark para la computación distribuida. Toda la arquitectura, compuesta por más de siete servicios independientes, se ha desplegado y orquestado mediante Docker y Docker Compose, garantizando la reproducibilidad y portabilidad del entorno.

El proyecto culmina con una prueba de concepto funcional que demuestra la viabilidad del pipeline: un vídeo grabado con Jitsi es consumido y procesado por un trabajo de Spark escrito en Python, que le aplica una transformación visual (un tinte de color de pantalla dividida) utilizando la librería OpenCV, validando así el flujo de datos completo.

Descriptores

Procesamiento de Vídeo, Big Data, Arquitectura de Microservicios, Apache Kafka, Apache Spark, Jitsi, Jibri, Docker, Docker Compose, Telerehabilitación, Visión Artificial, Pipeline de Datos.

Abstract

In the current technological context, video data generation has grown exponentially, creating the need to develop robust, scalable, and efficient systems for its processing. This project addresses this challenge by designing and implementing a complete, end-to-end software infrastructure, from video capture to its analysis.

The system uses Jitsi Meet as a videoconferencing platform and its Jibri component for recording sessions, generating the source video files. For ingestion and processing, a data pipeline has been deployed using Apache Kafka as a messaging broker in its modern KRaft mode and an Apache Spark cluster for distributed computing. The entire architecture, comprising more than seven independent services, has been deployed and orchestrated using Docker and Docker Compose, ensuring the reproducibility and portability of the environment.

The project culminates in a functional proof of concept that demonstrates the pipeline's viability: a video recorded with Jitsi is consumed and processed by a Spark job written in Python, which applies a visual transformation (a split-screen color tint) using the OpenCV library, thus validating the entire data flow.

Keywords

Video Processing, Big Data, Microservices Architecture, Apache Kafka, Apache Spark, Jitsi, Jibri, Docker, Docker Compose, Telerehabilitation, Computer Vision, Data Pipeline.

Índice general

Índice general	iii
Índice de figuras	v
Índice de tablas	vii
1 Introducción	1
2. Objetivos del proyecto	3
2.1. Objetivos generales	3
2.2. Objetivos técnicos	3
2.3. Objetivos personales	4
3 Conceptos Teóricos	7
3.1. Paradigmas de Arquitecturas Distribuidas	7
3.2. Fundamentos de Tecnologías de Streaming	8
3.3. Conceptos de Seguridad en Comunicaciones Web	10
3.4. Componentes Conceptuales de un Pipeline de Datos	11
4 Técnicas y herramientas	15
4.1. Herramientas de Desarrollo y Gestión	15
4.2. Infraestructura y Pipeline de Datos	17
5 Aspectos relevantes del desarrollo del proyecto	21
5.1. Fase Inicial: Diseño de la Arquitectura y Selección de Tecnologías	21
5.2. Fase I: El Desafío de Windows con WSL2	22
5.3. Fase II: Implementación en Debian	25
5.4. Fase III: Consolidación de la Infraestructura	29

6	Trabajos relacionados	35
6.1.	Proyecto de Origen: El Sistema FIS-FBIS	35
7	Conclusiones y Líneas de trabajo futuras	39
7.1.	Conclusiones	39
7.2.	Líneas de trabajo futuras	40
	Bibliografía	43

Índice de figuras

1.1. Ejemplo de la interfaz de Jitsi Meet, la plataforma de videoconferencia utilizada para las sesiones de tele-rehabilitación.	2
3.1. Arquitectura de red implementada para permitir el acceso externo seguro vía HTTPS y la validación de certificados de Let's Encrypt.	10
3.2. Visión general de la arquitectura del sistema, mostrando la interacción entre el subsistema de captura (Jitsi) y el pipeline de procesamiento (Kafka/Spark).	12
4.1. Ejemplo del tablero Kanban utilizado en Jira para el seguimiento de las tareas del Sprint.	16
4.2. Ejemplo del tablero Kanban utilizado en Jira para el seguimiento de las tareas del Sprint.	17
4.3. Contenedores levantados para videoconferencia Jitsi.	19
4.4. Contenedores de Spark y Kafka iniciándose	20
5.1. Ejemplo de algun error que surgio en esta fase wsl 2	23
5.2. Error de conexión inicial encontrado en el despliegue de Jitsi sobre Debian, a pesar de que los contenedores estaban en ejecución.	26
5.3. Panel de configuración de DuckDNS, donde se asocia un subdominio público a la dirección IP de la red local.	27
5.4. Ejemplo de la configuración de redirección de puertos (Port Forwarding) necesaria para Jitsi y Let's Encrypt.	28
5.5. Ejemplo de funcionamiento Let's Encrypt.	29
5.6. Captura de la sala de entrada a la reunión de Jitsi sin errores .	30
5.7. Captura donde se aprecia todos los contenedores funcionales y el script sin errores.	32

5.8. Captura donde se aprecia el video ya procesado mediante la transformación de color.	32
--	----

Índice de tablas

Capítulo 1

Introducción

La enfermedad de Parkinson es un trastorno neurodegenerativo que afecta a millones de personas en todo el mundo, con una incidencia especialmente alta en personas de edad avanzada [14]. Para estos pacientes, la rehabilitación física es clave para mantener su calidad de vida, pero el acceso a estas terapias a menudo supone un desafío considerable. Esta problemática se agrava en zonas rurales o despobladas, donde la distancia a los centros de salud y la dependencia de terceros para el desplazamiento complican o incluso impiden la continuidad del tratamiento, afectando negativamente a la evolución del paciente.

En este contexto, la telerehabilitación surge como una solución cada vez más necesaria para superar las dificultades de desplazamiento y la saturación de los servicios sanitarios [6]. Este Trabajo de Fin de Grado se enmarca en el proyecto de investigación **FIS PI19/00670** [1], en colaboración con el Hospital Universitario de Burgos (HUBU), que busca precisamente mejorar las terapias a distancia para estos pacientes.

El punto de partida de este trabajo es un sistema previo, desarrollado en el marco del proyecto mencionado, que ya permitía un análisis básico de los ejercicios grabados en vídeo. Sin embargo, su infraestructura tecnológica presentaba limitaciones de escalabilidad y fiabilidad que dificultaban su uso a gran escala. El objetivo principal de este TFG es, por tanto, el rediseño y la construcción de un *backend* mucho más robusto, capaz de procesar múltiples flujos de vídeo de forma eficiente y estable, sentando así las bases para un futuro sistema de *feedback* automático fiable.

Para conseguirlo, se ha diseñado e implementado una nueva arquitectura basada en los paradigmas de microservicios y procesamiento de datos a gran

escala. La solución implementa un *pipeline* de datos que orquesta la captura de las sesiones de vídeo, su transporte a través de un sistema de mensajería distribuida y su posterior análisis mediante un motor de computación en clúster. Toda esta infraestructura se ha empaquetado mediante tecnologías de contenerización para garantizar un despliegue consistente, reproducible y fácilmente escalable.

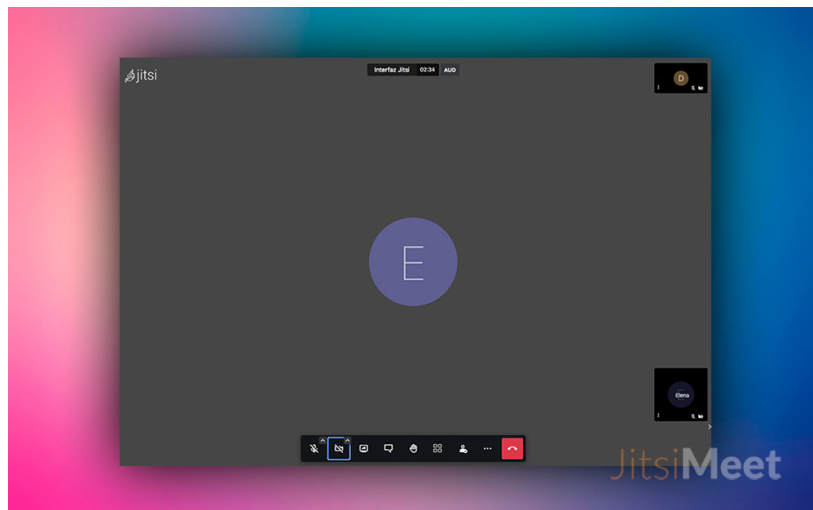


Figura 1.1: Ejemplo de la interfaz de Jitsi Meet, la plataforma de videoconferencia utilizada para las sesiones de tele-rehabilitación.

La presente memoria documenta todo el proceso de investigación, diseño y desarrollo llevado a cabo. El Capítulo ?? detalla los objetivos generales y técnicos del proyecto. En el Capítulo 3 se exponen los fundamentos teóricos de las tecnologías empleadas. El Capítulo ?? describe en profundidad las herramientas y la metodología de trabajo utilizadas. El Capítulo 5 narra los aspectos más relevantes del desarrollo, incluyendo los desafíos encontrados y las decisiones de diseño tomadas. Finalmente, el Capítulo 7 presenta las conclusiones extraídas del trabajo realizado y propone posibles líneas de trabajo futuras.

2. Objetivos del proyecto

En este capítulo se definen las metas y los objetivos que se han perseguido a lo largo del desarrollo de este Trabajo de Fin de Grado. Estos se han dividido en objetivos generales, que describen el propósito principal del proyecto, y objetivos técnicos, que detallan las metas específicas a nivel de implementación y tecnología. Finalmente, se incluye una sección con los objetivos personales que se han originado con la realización de este trabajo.

2.1. Objetivos generales

El objetivo principal de este proyecto es el diseño y la implementación de una infraestructura de *backend* para un sistema de procesamiento de vídeo destinado a la tele-rehabilitación de pacientes con la enfermedad de Parkinson.

Este objetivo general se fundamenta en la necesidad de evolucionar un sistema preexistente, asegurando que la nueva arquitectura pueda manejar de forma fiable la captura, el transporte y el almacenamiento de los flujos de datos de vídeo, sentando así las bases para futuras implementaciones de algoritmos de análisis y *feedback* automático.

2.2. Objetivos técnicos

Para alcanzar el objetivo general, se establecieron las siguientes metas técnicas específicas:

- Investigar y seleccionar un conjunto de tecnologías de código abierto adecuadas para construir un *pipeline* de datos en *streaming*, centrándose en sus licencias.
- Desplegar y configurar un sistema de captura de vídeo basado en **Jitsi**, utilizando su componente **Jibri** para la grabación de las sesiones de ejercicios en formato de archivo (MP4).
- Utilizar **Apache Kafka** como un bus de mensajería distribuido para desacoplar los componentes del sistema y garantizar un transporte de datos fiable.
- Integrar **Apache Spark** en la arquitectura como motor de procesamiento, preparando el sistema para el futuro análisis de los datos de vídeo almacenados o en *streaming*.
- Contenerizar toda la infraestructura utilizando **Docker** y **Docker Compose**, con el fin de crear un entorno de despliegue consistente, reproducible y fácil de gestionar.
- Documentar de manera exhaustiva todo el proceso de investigación, diseño, implementación y los desafíos técnicos encontrados, utilizando L^AT_EX a través de la plataforma Overleaf.
- Utilizar herramientas modernas de gestión de proyectos (**Jira**) y control de versiones (**Git/GitHub**) para llevar un seguimiento riguroso del trabajo realizado y asegurar la integridad del código fuente y la documentación.

2.3. Objetivos personales

Más allá de los requisitos técnicos, la realización de este TFG permite perseguir una serie de objetivos personales y de desarrollo profesional:

- Aplicar los conocimientos teóricos adquiridos a un problema de ingeniería complejo y real.
- Profundizar y ganar experiencia práctica en tecnologías de alta demanda en la industria, como son las plataformas (Kafka, Spark) y las herramientas (Docker).

- Afrontar y resolver problemas técnicos, desarrollando la capacidad de análisis, depuración y toma de decisiones ante imprevistos, como los surgidos durante el despliegue de la infraestructura.
- Contribuir, aunque sea a nivel de infraestructura, a un proyecto con un impacto social positivo y relevante.

Capítulo 3

Conceptos Teóricos

Para comprender las decisiones de diseño y la implementación de la infraestructura de este TFG, es necesario primero asentar las bases teóricas sobre las que se construyen las tecnologías seleccionadas. Este capítulo profundiza en los paradigmas y arquitecturas clave del procesamiento de datos distribuidos, el streaming de vídeo y los conceptos de red y seguridad que son el núcleo de este proyecto.

3.1. Paradigmas de Arquitecturas Distribuidas

Un sistema distribuido se compone de múltiples componentes de software autónomos, ejecutándose en diferentes nodos, que se comunican y coordinan entre sí para cumplir un objetivo común. El diseño de este tipo de sistemas se basa en una serie de patrones y paradigmas que garantizan su eficiencia, escalabilidad y tolerancia a fallos.

Sistemas de Mensajería: El Modelo Publicar-Suscribir

El modelo de comunicación Publicar-Suscribir (o *Pub-Sub*) es un patrón de mensajería asíncrono donde las aplicaciones que envían mensajes, llamadas **productores** (*publishers*), no los envían directamente a los receptores. En su lugar, los publican en canales o categorías lógicas, conocidas como **tópicos** (*topics*), gestionados por un intermediario o *broker*. Por otro lado, las aplicaciones que reciben los mensajes, llamadas **consumidores** (*subscribers*),

se suscriben a los tópicos de su interés para recibir los mensajes que se publican en ellos [5].

La principal ventaja de este patrón es el **desacoplamiento** que introduce entre los componentes. Los productores no necesitan conocer a los consumidores, y viceversa. Esto permite que los sistemas escalen de forma independiente y que nuevos componentes puedan añadirse a la arquitectura sin necesidad de modificar los existentes, dota a la arquitectura de una enorme flexibilidad y escalabilidad [12].

Virtualización a Nivel de Sistema Operativo

La virtualización es una técnica que permite crear versiones virtuales de recursos informáticos. Mientras que las máquinas virtuales tradicionales emulan un sistema de hardware completo sobre el que se ejecuta un sistema operativo invitado, la **virtualización a nivel de sistema operativo**, más conocida como contenerización, sigue un enfoque diferente.

Los contenedores se ejecutan sobre el kernel del sistema operativo anfitrión, pero en espacios de usuario aislados. Cada contenedor tiene su propio sistema de ficheros, procesos y configuración de red, pero comparte el mismo kernel subyacente. Esto los hace extremadamente ligeros, rápidos de iniciar y mucho más eficientes en el uso de recursos que las máquinas virtuales, convirtiéndolos en la tecnología ideal para desplegar arquitecturas de microservicios.

3.2. Fundamentos de Tecnologías de Streaming

El Log de Transacciones Distribuido e Inmutable

Una de las arquitecturas más potentes para construir sistemas de mensajería y plataformas de *streaming* de datos es el **log de transacciones distribuido** (o *commit log*). Conceptualmente, es una estructura de datos inmutable a la que solo se pueden añadir registros. Una vez escrito, un registro no puede ser modificado ni eliminado.

En un sistema distribuido, este log se divide en **particiones** para permitir el paralelismo y la escalabilidad. Cada partición es un log ordenado que se replica a través de múltiples servidores (**brokers**) para garantizar la tolerancia a fallos y la alta disponibilidad. Cada registro en una partición

tiene un identificador secuencial único llamado **offset**, que permite a los consumidores llevar un control preciso de su posición de lectura, incluso en caso de fallos. Este modelo es la base de sistemas de mensajería de alto rendimiento capaces de procesar millones de eventos por segundo.

Modelo de Computación Distribuida en Memoria

El procesamiento de grandes volúmenes de datos (*Big Data*) requiere modelos de computación que puedan paralelizar el trabajo a través de un clúster de máquinas. La abstracción fundamental en los motores de procesamiento modernos es el **Conjunto de Datos Distribuido y Resiliente** (*Resilient Distributed Dataset* o RDD). Un RDD es una colección de elementos inmutable y tolerante a fallos que puede ser procesada en paralelo [19].

Las operaciones sobre estos datos se dividen en:

- **Transformaciones:** Operaciones que crean un nuevo conjunto de datos a partir de uno existente (ej. un filtrado). Son "perezosas" (*lazy*), es decir, su cómputo no se realiza al momento.
- **Acciones:** Operaciones que disparan la ejecución de las transformaciones y devuelven un resultado o escriben en un sistema externo.

Cuando se invoca una acción, el motor de procesamiento analiza la cadena de transformaciones y construye un **Grafo Acíclico Dirigido (DAG)** de las operaciones. Este grafo es optimizado y dividido en etapas de tareas que se distribuyen entre los nodos del clúster para su ejecución paralela, permitiendo un procesamiento masivo y eficiente.

Transmisión de Vídeo sobre IP

La transmisión de vídeo en tiempo real a través de redes IP se basa en protocolos diseñados para manejar datos multimedia con baja latencia. Uno de los protocolos más extendidos para este fin es el **Protocolo de Mensajería en Tiempo Real (RTMP)**. Desarrollado originalmente para Adobe Flash, RTMP es un protocolo basado en TCP que mantiene una sesión persistente entre cliente y servidor, permitiendo la transmisión fiable de flujos de audio, vídeo y datos.

3.3. Conceptos de Seguridad en Comunicaciones Web

HTTP, HTTPS y el Protocolo SSL/TLS

HTTP (Hypertext Transfer Protocol) es el protocolo fundamental de la web, pero transmite la información en texto plano. **HTTPS** es su versión segura, que añade una capa de cifrado mediante el protocolo **SSL/TLS** (Secure Sockets Layer/Transport Layer Security) [17]. Esta capa utiliza criptografía de clave pública para establecer una conexión segura, garantizando la confidencialidad (nadie puede leer la comunicación) y la integridad (nadie puede modificarla) de los datos.

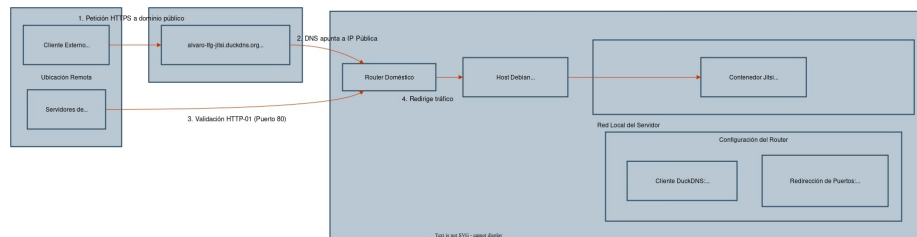


Figura 3.1: Arquitectura de red implementada para permitir el acceso externo seguro vía HTTPS y la validación de certificados de Let's Encrypt.

Certificados Digitales y Autoridades de Certificación

Para que HTTPS funcione, el servidor debe presentar un **certificado digital SSL/TLS** que valide su identidad. Existen dos tipos principales:

- **Certificados Autofirmados:** Son generados por el propio administrador del servidor. No son emitidos por una entidad de confianza, por lo que los navegadores web los rechazan por defecto, mostrando advertencias de seguridad al usuario.
- **Certificados de una CA:** Son emitidos por una **Autoridad de Certificación (CA)**, una entidad externa en la que confían los navegadores (como Let's Encrypt). Cuando un navegador recibe un certificado emitido por una CA de confianza, establece la conexión segura sin advertencias. Por motivos de seguridad, los navegadores modernos exigen una conexión HTTPS con un certificado válido para permitir el acceso a funcionalidades sensibles como la cámara o el micrófono.

3.4. Componentes Conceptuales de un Pipeline de Datos

Toda arquitectura de procesamiento de datos, como la que se aborda en este proyecto, puede descomponerse en una serie de componentes lógicos, cada uno con una responsabilidad bien definida. La comprensión de estos roles es fundamental para diseñar un sistema robusto y escalable.

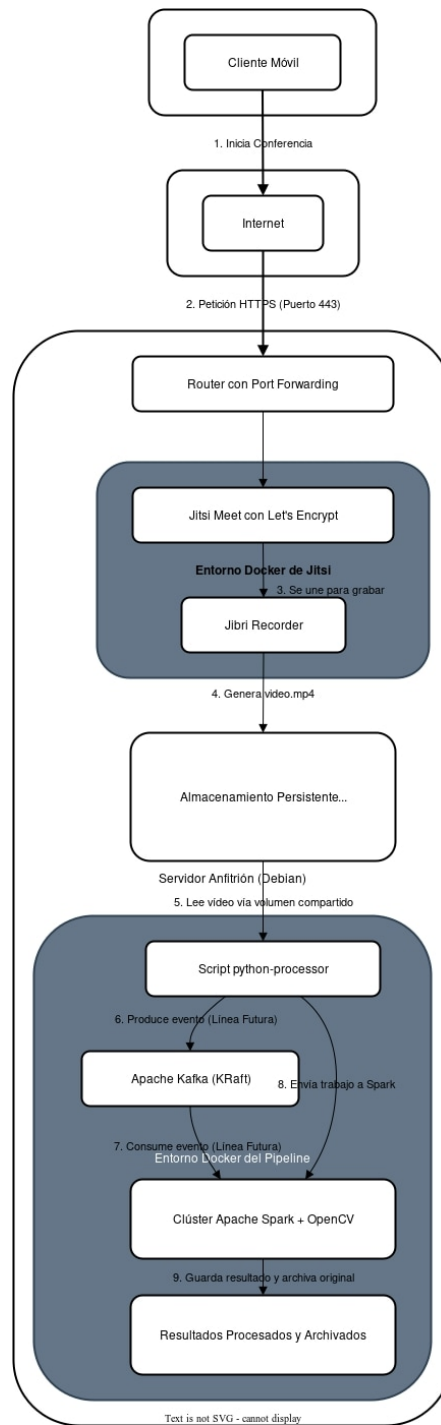


Figura 3.2: Visión general de la arquitectura del sistema, mostrando la interacción entre el subsistema de captura (Jitsi) y el pipeline de procesamiento (Kafka/Spark).

3.4. COMPONENTES CONCEPTUALES DE UN PIPELINE DE DATOS

El Productor de Eventos (*Producer*)

El **Productor** es cualquier entidad del sistema cuya función principal es originar o capturar datos y enviarlos a un sistema de mensajería. Este componente actúa como la fuente de información del pipeline. En el contexto de un sistema de análisis de vídeo, el rol del productor sería desempeñado por el sistema de captura, encargado de digitalizar la sesión y publicarla como un evento o una serie de eventos en el flujo de datos.

El Intermediario de Mensajería (*Broker*)

El **Broker** es el servidor o conjunto de servidores que forman el núcleo del sistema de mensajería. Su responsabilidad es recibir los eventos de los productores, almacenarlos de forma duradera y fiable, y ponerlos a disposición de los consumidores. En arquitecturas distribuidas, se despliega un clúster de brokers para garantizar la alta disponibilidad y la tolerancia a fallos. Este componente es esencial para desacoplar a los productores de los consumidores, permitiendo que operen a ritmos diferentes y de forma independiente.

El Consumidor y Procesador de Datos (*Consumer/Processor*)

El **Consumidor** es la aplicación que se suscribe al sistema de mensajería para recibir y leer los eventos que le interesan. Una vez que un consumidor recibe un evento, se lo entrega a una lógica de **Procesamiento**, que es donde reside la inteligencia de la aplicación. Esta lógica es la que se encarga de transformar, analizar o actuar sobre los datos recibidos. En un sistema de procesamiento de vídeo, el consumidor se encargaría de recibir los datos del vídeo, y el procesador aplicaría los algoritmos de visión artificial sobre ellos.

El Modelo de Computación Maestro-Trabajador (*Master-Worker*)

Para procesar grandes volúmenes de datos de manera eficiente, los *frameworks* de computación distribuida suelen emplear el patrón arquitectónico **Maestro-Trabajador** (conocido también como *Driver-Executor*).

- El nodo **Maestro** (*Master* o *Driver*) es el cerebro de la operación. Recibe el trabajo a realizar, lo divide en un conjunto de tareas más pequeñas e independientes, y las distribuye entre los nodos trabajadores.

También se encarga de coordinar la ejecución y agregar los resultados finales.

- Los nodos **Trabajadores** (*Workers* o *Executors*) son los que realizan el trabajo pesado. Cada trabajador recibe una o más tareas del maestro, las ejecuta en paralelo con otros trabajadores sobre una porción de los datos, y devuelve el resultado al maestro.

Capítulo 4

Técnicas y herramientas

En este capítulo se describen en detalle las principales tecnologías, lenguajes de programación y herramientas que han constituido la base para el desarrollo de este Trabajo de Fin de Grado. El objetivo de esta sección es presentar cada componente de forma objetiva, explicando su función, características y arquitectura principal.

4.1. Herramientas de Desarrollo y Gestión

Python

Python es un lenguaje de programación de alto nivel, interpretado y multiparadigma, cuya filosofía de diseño enfatiza la legibilidad del código [15]. Es mantenido por la Python Software Foundation y cuenta con una vasta biblioteca estándar y un ecosistema de librerías de terceros que lo han convertido en un estándar de facto en el desarrollo de aplicaciones, la ciencia de datos y el *machine learning*.

Jira y Metodología Ágil

Jira es una herramienta de software desarrollada por Atlassian para el seguimiento de incidencias y la gestión de proyectos [4]. Está diseñada para dar soporte a metodologías de desarrollo ágil como Scrum, que se basa en la organización del trabajo en iteraciones cortas denominadas Sprints. Jira facilita la gestión de un *backlog* de tareas, la planificación de Sprints y la visualización del flujo de trabajo mediante tableros Kanban.

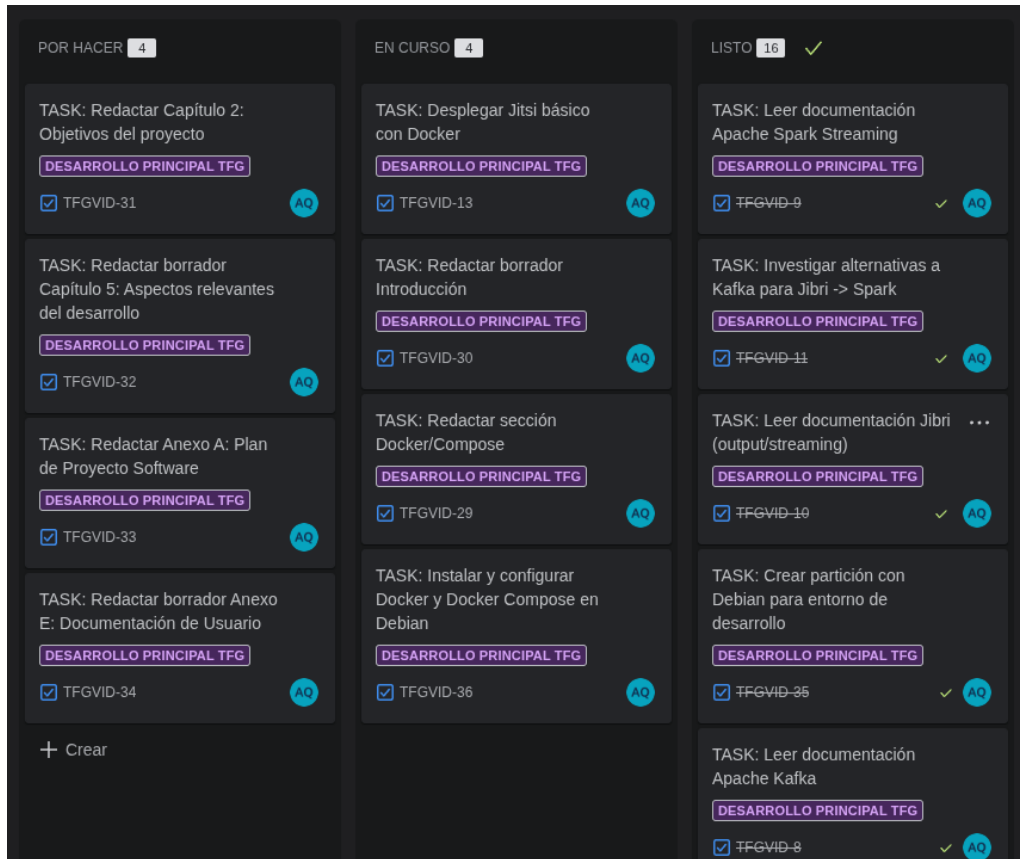


Figura 4.1: Ejemplo del tablero Kanban utilizado en Jira para el seguimiento de las tareas del Sprint.

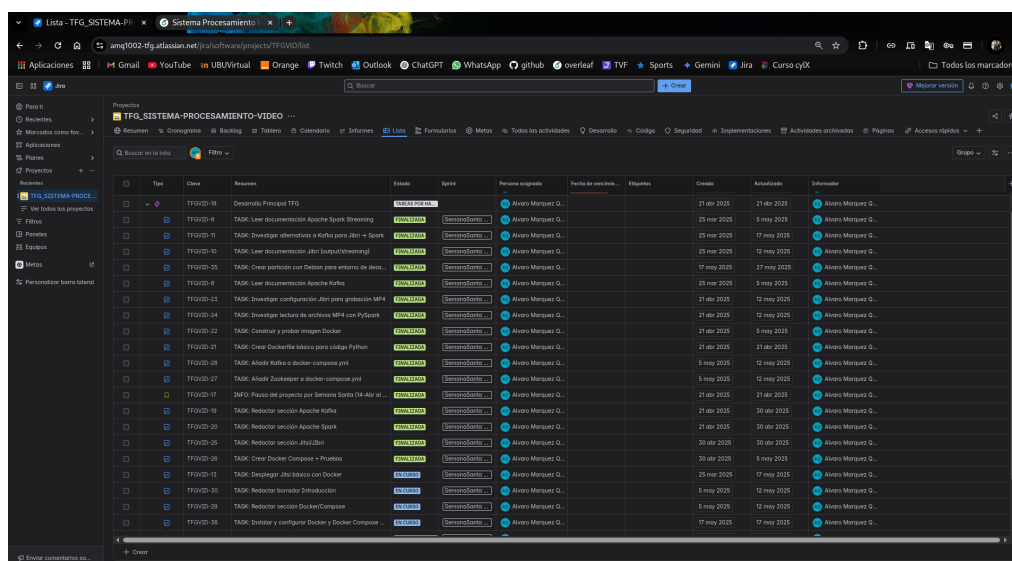


Figura 4.2: Ejemplo del tablero Kanban utilizado en Jira para el seguimiento de las tareas del Sprint.

4.2. Infraestructura y Pipeline de Datos

El núcleo del proyecto reside en la infraestructura diseñada para capturar, transportar y procesar los datos de vídeo.

Contenerización con Docker y Docker Compose

Docker es una plataforma de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores [7]. Un contenedor es una unidad de software estándar, ligera y ejecutable que incluye todo lo necesario para que una aplicación se ejecute: código, *runtime*, herramientas del sistema y librerías. A diferencia de las máquinas virtuales, los contenedores comparten el kernel del sistema operativo anfitrión, lo que los hace mucho más eficientes en el uso de recursos. La gestión de contenedores se realiza a través de un **Dockerfile**, un fichero de texto que contiene las instrucciones para construir una imagen de contenedor.

Para la orquestación de los múltiples contenedores de este proyecto, se utiliza **Docker Compose**. Esta herramienta permite definir y gestionar aplicaciones multi-contenedor a través de un único archivo de configuración en formato YAML (`docker-compose.yml`). En este fichero se definen los

diferentes servicios que componen la aplicación, sus imágenes, las redes que los conectan, los volúmenes de datos para la persistencia y las dependencias de arranque entre ellos.

Plataforma de Videoconferencia: Jitsi

Jitsi es una colección de proyectos de software libre para construir soluciones de videoconferencia seguras y escalables [10]. Su arquitectura es distribuida y se compone de varios microservicios que trabajan de forma coordinada:

- **Jitsi Meet:** Es la aplicación cliente, una interfaz web basada en JavaScript y WebRTC que se ejecuta en el navegador del usuario y le permite participar en las conferencias.
- **Jitsi Videobridge (JVB):** Es el componente central. Funciona como una Unidad de Reenvío Selectivo (SFU), lo que significa que recibe los flujos de vídeo de cada participante y los reenvía de forma selectiva al resto, sin necesidad de decodificarlos y mezclarlos. Esta arquitectura es clave para la eficiencia y escalabilidad del sistema.
- **Prosody:** Es el servidor *XMPP* que gestiona la señalización en las conferencias: quién entra o sale de una sala, el estado de los participantes, los mensajes de chat, etc.
- **Jicofo (Jitsi Conference Focus):** Actúa como el "director de orquesta" de las conferencias. Es el componente que gestiona las sesiones, invita a los nuevos participantes (incluido Jibri) a la conferencia y asigna los flujos de vídeo al Jitsi Videobridge.
- **Jibri (Jitsi Broadcasting Infrastructure):** Es el componente utilizado en este TFG para la captura de datos. Jibri es un servicio que se une a una conferencia como un participante silencioso, lanza una instancia de un navegador Chrome en un servidor virtual (*headless*) y utiliza FFmpeg para capturar y grabar la vista compuesta de la conferencia en un archivo de vídeo estándar (formato MP4).


```

[+] Running 3/3
  jibri Pulled                                23.1s
  7991c2430a8 Pull complete                  21.2s
  5cfff3a8b6a44 Pull complete                0.5s
[+] Running 6/6
  Network docker-jitsi-meet-meet-jitsi Created                                0.0s
  Container docker-jitsi-meet-prosody-1 Started                             1.8s
  Container docker-jitsi-meet-jvb-1 Started                                0.9s
  Container docker-jitsi-meet-jicofo-1 Started                             0.8s
  Container docker-jitsi-meet-web-1 Started                               1.0s
  Container docker-jitsi-meet-jibri-1 Started                               0.9s

jgonzalez@MARQUEZ-PC MINGW64 /g/Otros ordenadores/Mi portátil/Escritorio/CURSO 24-25/TFG_VIDEO_PROCES/TFG-SISTEMA-PROCESAMIENTO-VIDEO/HerramientasExternas/docker-jitsi-meet (master)
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
50de32f40ce   jitsi/jibri:unstable "/init"               About a minute ago Up About a minute                               docker-jitsi-meet-jibri-1
e7d9316eecec   jitsi/web:unstable "/init"               About a minute ago Up About a minute   0.0.0.0:8000->80/tcp, 0.0.0.0:8443->443/tcp  docker-jitsi-meet-web-1
8969d24b5b5   jitsi/jicofo:unstable "/init"               About a minute ago Up About a minute   127.0.0.1:8888->8888/tcp                    docker-jitsi-meet-jicofo-1
0632cf3335    jitsi/jvb:unstable "/init"               About a minute ago Up About a minute   227.0.0.1:8080->8080/tcp, 0.0.0.0:10000->10000/udp  docker-jitsi-meet-jvb-1
4e0db318a792   jitsi/prosody:unstable "/init"               About a minute ago Up About a minute   5222/tcp, 5269/tcp, 5280/tcp, 5347/tcp        docker-jitsi-meet-prosody-1

```

Figura 4.3: Contenedores levantados para videoconferencia Jitsi.

Plataformas de Big Data: Kafka y Spark

Apache Kafka: Un Log de Transacciones Distribuido

Apache Kafka es una plataforma distribuida de código abierto para la transmisión de eventos (*event streaming*) [2]. Aunque a menudo se utiliza como un sistema de mensajería publicar-suscribir, su arquitectura fundamental es la de un **log de transacciones distribuido, particionado y replicado**. Los conceptos clave de su arquitectura son:

- **Brokers y Clúster:** Kafka se ejecuta como un clúster de uno o más servidores, denominados *brokers*. Estos brokers gestionan el almacenamiento y la atención a las peticiones de los clientes.
- **Topics y Particiones:** Los flujos de eventos se organizan en categorías llamadas *topics*. Para permitir el paralelismo y la escalabilidad, cada topic se divide en una o más *particiones*. Cada partición es un log ordenado e inmutable al que solo se pueden añadir nuevos registros.
- **Productores y Consumidores:** Las aplicaciones que escriben datos en los topics de Kafka se denominan *Productores*. Las que leen los datos son los *Consumidores*, que se organizan en grupos para procesar los datos de forma paralela.
- **Modo KRaft (sin Zookeeper):** Las versiones modernas de Kafka, como la utilizada en este proyecto, pueden operar en modo KRaft (Kafka Raft). En este modo, Kafka utiliza un protocolo de consenso interno para gestionar los metadatos del clúster, eliminando la necesidad de un sistema de coordinación externo como Apache Zookeeper y simplificando así significativamente la arquitectura de despliegue.

```
[*] Running 6/6
✓ python-processor Built 0.0s
✓ Network src_processing_net Created 0.0s
✓ Container src-kafka-1 Created 0.0s
✓ Container src-spark-master-1 Created 0.0s
✓ Container src-spark-worker-1 Created 0.0s
✓ Container src-python-processor-1 Created 0.0s
Attaching to kafka-1, python-processor-1, spark-master-1, spark-worker-1
spark-master-1 | spark 14:48:02.35 INFO ==>
spark-master-1 | spark 14:48:02.35 INFO ==> Welcome to the Bitnami spark container
spark-master-1 | spark 14:48:02.35 INFO ==> Subscribe to project updates by watching https://github.com/bitnami/containers
spark-master-1 | spark 14:48:02.35 INFO ==> Did you know there are enterprise versions of the Bitnami catalog? For enhanced secure software supply chain features, unlimited pulls from Docker, L
TS support, or application customization, see Bitnami Premium or Tanzu Application Catalog. See https://www.arrow.com/globalecs/na/vendors/bitnami/ for more information.
spark-master-1 | spark 14:48:02.35 INFO ==>
kafka-1 | kafka 14:48:02.36 INFO ==>
kafka-1 | kafka 14:48:02.36 INFO ==> Welcome to the Bitnami kafka container
kafka-1 | kafka 14:48:02.36 INFO ==> Subscribe to project updates by watching https://github.com/bitnami/containers
kafka-1 | kafka 14:48:02.37 INFO ==> Did you know there are enterprise versions of the Bitnami catalog? For enhanced secure software supply chain features, unlimited pulls from Docker, L
TS support, or application customization, see Bitnami Premium or Tanzu Application Catalog. See https://www.arrow.com/globalecs/na/vendors/bitnami/ for more information.
kafka-1 | kafka 14:48:02.37 INFO ==>
kafka-1 | kafka 14:48:02.37 INFO ==> ** Starting Kafka setup **
spark-worker-1 | spark 14:48:02.48 INFO ==>
spark-worker-1 | spark 14:48:02.48 INFO ==> Welcome to the Bitnami spark container
spark-worker-1 | spark 14:48:02.48 INFO ==> Subscribe to project updates by watching https://github.com/bitnami/containers
spark-worker-1 | spark 14:48:02.48 INFO ==> Did you know there are enterprise versions of the Bitnami catalog? For enhanced secure software supply chain features, unlimited pulls from Docker, L
TS support, or application customization, see Bitnami Premium or Tanzu Application Catalog. See https://www.arrow.com/globalecs/na/vendors/bitnami/ for more information.
spark-worker-1 | spark 14:48:02.48 INFO ==>
```

Figura 4.4: Contenedores de Spark y Kafka iniciandose

Apache Spark: Un Motor de Computación Distribuida

Apache Spark es un motor de análisis unificado para el procesamiento de datos a gran escala [3]. Su principal característica es su capacidad para realizar computación distribuida en memoria, lo que le permite alcanzar un alto rendimiento. Su arquitectura se compone de:

- **Driver y Executors:** Una aplicación Spark consiste en un proceso *driver* que coordina la ejecución y múltiples procesos *executor* que se ejecutan en los nodos de un clúster. El driver analiza el código, planifica las tareas y las envía a los *executors* para que las ejecuten en paralelo sobre los datos.
- **DataFrames y el Optimizador Catalyst:** La API principal de Spark es el DataFrame, una abstracción de datos distribuidos organizados en columnas con nombre. Cuando se ejecutan operaciones sobre un *DataFrame*, el optimizador de Spark, llamado **Catalyst**, traduce estas operaciones a un plan lógico, lo optimiza y genera un plan físico de ejecución que se distribuye por el clúster.
- **PySpark:** Es la API de Python para Spark, que permite a los desarrolladores utilizar la potencia de Spark con la simplicidad y el amplio ecosistema de librerías de Python.

Capítulo 5

Aspectos relevantes del desarrollo del proyecto

En este capítulo se expone la crónica del desarrollo de este TFG. Lejos de ser un proceso lineal, el trabajo ha seguido un camino iterativo, marcado por la investigación, la implementación de prototipos, la aparición de desafíos técnicos y la toma de decisiones estratégicas para superar los bloqueos. Esta narrativa busca reflejar fielmente el proceso de ingeniería real que he llevado a cabo, detallando no solo los éxitos, sino también los problemas y el proceso de depuración que ha sido necesario para resolverlos.

5.1. Fase Inicial: Diseño de la Arquitectura y Selección de Tecnologías

El proyecto comenzó con una fase de investigación y planificación, documentada en el Anexo apéndice: plan de proyecto. Durante esta etapa, se tomaron decisiones fundamentales sobre la pila tecnológica y el alcance inicial del proyecto.

Selección de la Pila Tecnológica

Basándome en los requisitos de construir un sistema de procesamiento de vídeo escalable, se realizó un estudio de las tecnologías de Big Data más adecuadas, como se detalla en el Capítulo 4. La pila tecnológica seleccionada fue:

- **Jitsi/Jibri:** Como solución de código abierto para la captura de las sesiones de vídeo. Se ha optado por utilizar el proyecto oficial `docker-jitsi-meet` [18], ya que proporciona una configuración pre-empaquetada con Docker Compose que, teóricamente, simplifica el despliegue de todos sus micro servicios (web, prosody, jicofo, jvb, jibri).
- **Apache Kafka:** Como bus de mensajería para desacoplar la captura del procesamiento.
- **Apache Spark:** Como motor de procesamiento distribuido para el futuro análisis de los datos.

Definición del Alcance: Enfoque *Offline* como Primer Hito

Desde el principio, se conocía la complejidad de un sistema de procesamiento en tiempo real. La integración de un flujo RTMP de Jibri con Kafka y Spark Streaming presenta desafíos técnicos significativos. Por ello, se tomo una decisión estratégica clave para mitigar riesgos: **priorizar un flujo de trabajo *offline* como primer objetivo funcional**. En este enfoque, Jibri grabaría las sesiones en archivos MP4, que se almacenarían en un sistema de ficheros accesible para que Spark los procesara por lotes. Esto me permitió dividir el problema en fases manejables, asegurando la entrega de una prueba de concepto funcional antes de abordar la complejidad del streaming en tiempo real.

5.2. Fase I: El Desafío de Windows con WSL2

El primer intento de despliegue se realizó sobre el sistema de desarrollo principal: un sistema Windows 11 con el Subsistema de Windows para Linux (WSL2) y Docker Desktop. Aunque este entorno es muy versátil, la interacción entre el sistema de archivos de Windows (NTFS) y los permisos de Linux dentro de los contenedores demostró ser la fuente de problemas complejos y bloqueantes.

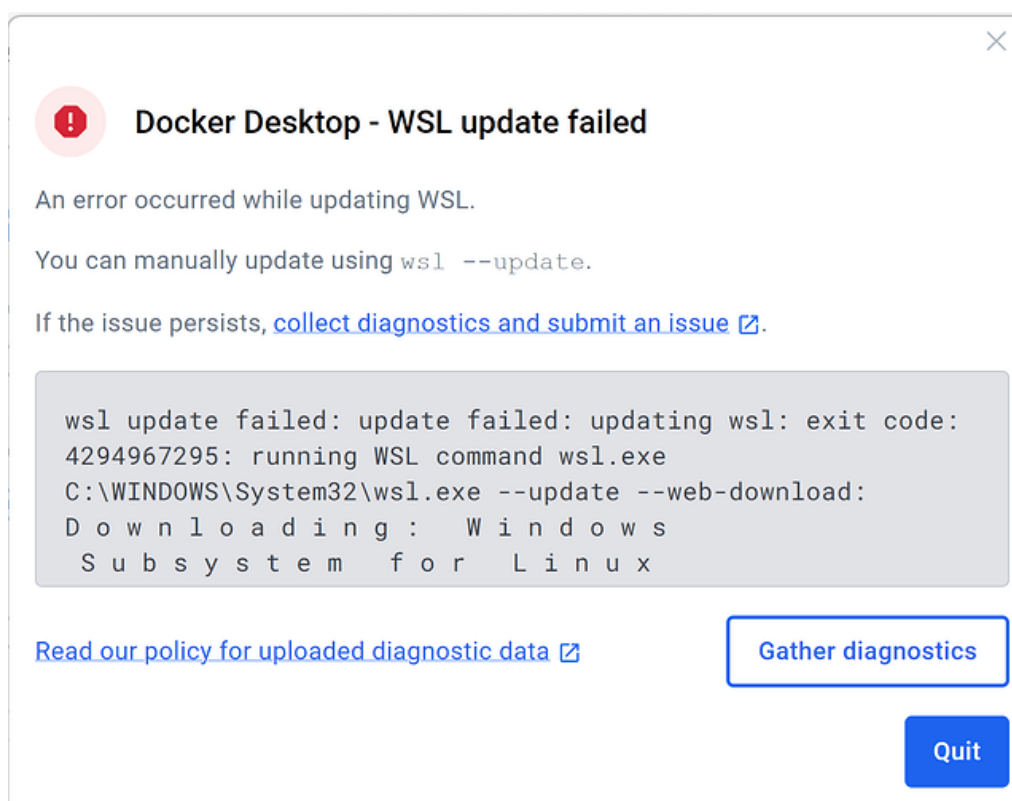


Figura 5.1: Ejemplo de algun error que surgio en esta fase wsl 2

Primer Despliegue de Jitsi y Análisis de Errores Críticos

Tras clonar el repositorio de `docker-jitsi-meet` y configurar el fichero `.env`, el primer intento de levantar la pila de servicios con `docker-compose up` resultó en un fallo en cascada. Dos componentes clave, Prosody y Jibri, fallaban sistemáticamente.

Errores de Permisos en el Contenedor Prosody

El análisis de los *logs* del contenedor de Prosody (el servidor XMPP) fue el primer indicio claro del problema subyacente. Los registros mostraban de forma repetida y consistente errores de tipo "Permission denied", como se puede observar en este extracto:

```
[frame=single, label={Fragmento de log de error de Prosody}]
```

```
datamanager error Unable to write to accounts storage  
('/config/data/auth%2elocalhost/accounts/focus.dat~:  
Permission denied')
```

Este error se debe a un conflicto en la gestión de permisos de ficheros entre sistemas operativos. El proceso de Prosody se ejecuta dentro del contenedor con un usuario y grupo específicos de Linux (*uid/gid*), pero el volumen donde intenta escribir (*/config/data*) está montado desde el sistema de archivos NTFS de Windows a través de WSL2. La capa de traducción de permisos de WSL2 no era capaz de mapear correctamente los permisos, impidiendo que el proceso tuviera los privilegios de escritura necesarios. A pesar de intentar modificar los permisos en Windows y probar diferentes configuraciones de montaje de volúmenes en Docker, el problema persistió, indicando que se trataba de una limitación fundamental del entorno.

Fallos de Autenticación en Cascada: Jibri y Jicofo

El fallo de Prosody provocó un efecto dominó. **Jicofo**, el componente que gestiona las conferencias, no podía establecer conexión con el servidor XMPP. De forma aún más crítica, **Jibri**, el grabador de vídeo, fallaba con dos errores distintos:

- Un error fatal **FATAL ERROR: Jibri recorder password and auth password must be set**, sugiriendo un problema en cómo se leían las variables de entorno desde el fichero `.env` en el entorno WSL2.
- Un error de autenticación: **SASLError using SCRAM-SHA-1: malformed-request**, consecuencia directa de no poder conectar con un servidor Prosody que no había logrado arrancar correctamente.

Decisión Estratégica: Pivote a un Entorno Linux Nativo

Tras un extenso periodo de depuración, se diagnosticó que los problemas de despliegue no residían en la configuración de Jitsi, sino en incompatibilidades fundamentales de la infraestructura de desarrollo basada en *Windows* y *WSL2*. La resolución de estos conflictos de bajo nivel, relacionados con la gestión de permisos y redes, se consideró un riesgo que desviaba el foco de los objetivos principales del proyecto.

En consecuencia, se tomó la decisión estratégica de migrar el entorno de desarrollo a un sistema operativo *Linux* nativo. Se seleccionó *Debian* debido

a su reconocida estabilidad en entornos de servidor y su capacidad para proporcionar un entorno de ejecución predecible, eliminando las capas de compatibilidad que originaban los errores.

5.3. Fase II: Implementación en Debian

La migración a un entorno Linux nativo fue un punto de inflexión en el proyecto. Como esperaba, los problemas de permisos relacionados con WSL2 desaparecieron de inmediato. Al ejecutar el comando `docker-compose up -d` en el directorio de `docker-jitsi-meet`, los cinco contenedores principales (prosody, jicofo, jvb, web y jibri) lograron arrancar y mantenerse en ejecución, un hito que no había sido posible alcanzar anteriormente.

Sin embargo, el éxito inicial solo reveló la siguiente capa de desafíos. A pesar de que los servicios estaban activos, la interfaz web de Jitsi no era accesible, mostrando un error de *"La conexión ha fallado"*. Esto dio comienzo a un nuevo e intensivo ciclo de depuración, esta vez centrado en la configuración de la red y los componentes internos de Jitsi.

Proceso de Depuración de la Conexión HTTP

Para solucionar el problema de conexión, se realizó un proceso de diagnóstico sistemático, analizando los *logs* de cada contenedor y el comportamiento de la aplicación web para aislar la causa raíz.

Diagnóstico del Protocolo de Conexión (WSS)

El primer hallazgo importante se obtuvo al inspeccionar la consola de desarrollador del navegador. Esto reveló que, a pesar de que la instancia estaba configurada para HTTP, el cliente web intentaba establecer una conexión segura mediante el protocolo WebSocket Secure (WSS). Esto fallaba porque el servidor no tenía configurado ningún certificado SSL/TLS. La solución pasaba por forzar al cliente a utilizar una conexión no segura.

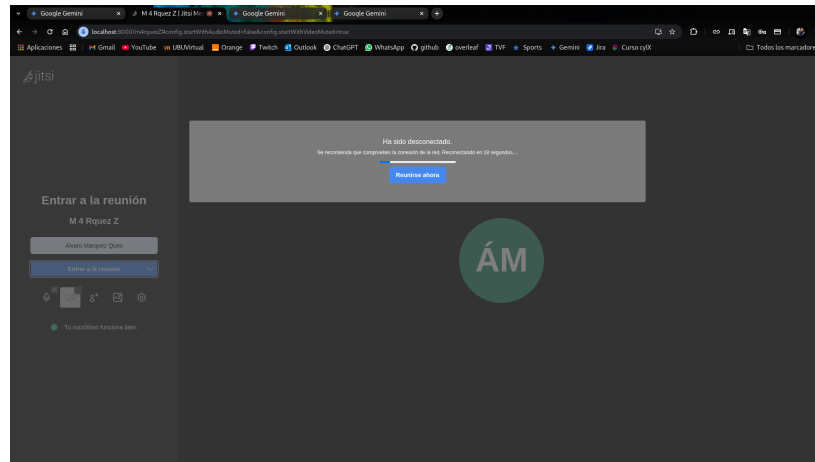


Figura 5.2: Error de conexión inicial encontrado en el despliegue de Jitsi sobre Debian, a pesar de que los contenedores estaban en ejecución.

Solución y Error de Montaje: `custom.config.js`

La documentación de Jitsi indica que se puede sobrescribir la configuración del cliente web mediante un archivo `custom.config.js`. Se decidió crear este archivo con el objetivo de deshabilitar la conexión segura. Sin embargo, al volver a lanzar los contenedores, el servicio web falló con un nuevo error: `...not a directory`. El problema era que Docker, al intentar montar el archivo de configuración en un volumen, se encontraba con que el directorio de destino ya había sido creado por el script de inicio del contenedor. La solución fue asegurar la creación de la estructura de directorios y el fichero `custom.config.js` en la carpeta de configuración del host (`~/.jitsi-meet-cfg/web/`) antes de ejecutar `docker-compose up` por primera vez.

Autenticación de Usuarios Anónimos (Invitados)

Una vez solucionado el problema anterior, se logró acceder a la interfaz web, pero esto no hizo más que generar un nuevo bloqueo: no se pudo crear una sala como usuario anónimo o invitado. Los `logs` de Prosody indicaron que la autenticación para el dominio de invitados no estaba configurada. Para solucionarlo, se decidió que editar manualmente el fichero de configuración principal de Prosody (`prosody.cfg.lua`) y añadir un nuevo `VirtualHost` para `"guest.jitsi.localhost"`, habilitando explícitamente la autenticación anónima. Tras este cambio, finalmente se consiguió crear una conferencia y establecer una comunicación de vídeo y audio funcional.

Iteración hacia HTTPS y Estrategia Actual

Con el sistema funcionando en HTTP, y siguiendo las recomendaciones de los tutores, el siguiente paso fue intentar una implementación con HTTPS.

- **Intento con Certificados Autofirmados:** El primer enfoque fue utilizar certificados autofirmados generados con la herramienta `mkcert`. Sin embargo, esto generó retroceder a un problema similar al de WSL2: el contenedor web de Jitsi (Nginx) no era capaz de leer o utilizar correctamente los certificados montados, provocando errores de SSL en el navegador.
- **Diagnóstico con OpenSSL:** Para diagnosticar el problema de forma precisa, se utilizó la herramienta de línea de comandos `openssl s_client` para conectarme directamente al puerto 443 del servidor. El análisis confirmó que el servidor no estaba presentando el certificado que le había proporcionado, sino uno por defecto, lo que indicaba un fallo en la configuración de Nginx o en cómo Docker montaba los certificados. La herramienta `openssl s_client` fue fundamental para este diagnóstico [13].

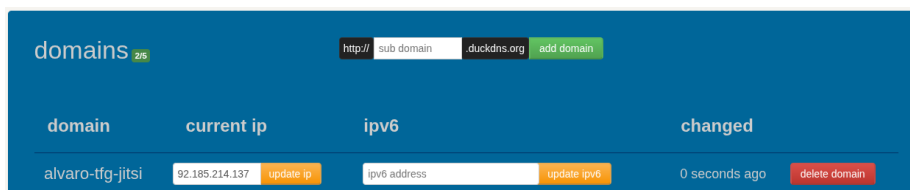



Figura 5.3: Panel de configuración de DuckDNS, donde se asocia un subdominio público a la dirección IP de la red local.

DHCP	NAT/PAT	DNS	UPnP	DynDNS	DMZ	NTP	ONT
------	---------	-----	------	--------	-----	-----	-----

Configuración de NAT/PAT/CGNAT

Estas normas son necesarias para autorizar una conexión remota desde Internet que llegue a un dispositivo específico de tu red LAN. También puedes definir los puertos(s) que utilizará esta comunicación.

Para crear la regla NAT debes introducir la IPv4 asignada a tu dispositivo en la LAN. Para saber cuál es puedes consultar el listado de IPs asignadas en la pestaña "DHCP" de esta página.

 Atención: Asegúrate de que no has filtrado estos puertos en el firewall.

Personalizar reglas						
estado	aplicación / servicio	puerto interno	puerto externo	protocolo	IPv4 del dispositivo	
	FTP Server	21	21	TCP	HPF233(192.168.1.1)	añadir
✓	Web Server (HTTP)	80	80	TCP	192.168.1.18	editar borrar
✓	Secure Web Server (HTTPS)	443	443	TCP	192.168.1.18	editar borrar

Figura 5.4: Ejemplo de la configuración de redirección de puertos (Port Forwarding) necesaria para Jitsi y Let's Encrypt.

- **Estrategia Definitiva (Let's Encrypt):** Basado en esta experiencia y en las indicaciones de los tutores, la estrategia actual y definitiva del proyecto es implementar HTTPS utilizando certificados válidos emitidos por una autoridad de certificación reconocida como **Let's Encrypt**. Este enfoque, aunque requiere un dominio público, garantiza una solución estándar, robusta y confiable, eliminando todos los problemas de confianza del navegador. Las tareas técnicas asociadas a esta implementación forman parte de la fase final del proyecto.

CAPÍTULO 5. ASPECTOS RELEVANTES DEL DESARROLLO DEL PROYECTO

Una vez configurado el entorno de red, bastó con habilitar las variables correspondientes en el fichero `.env` de Jitsi. Al levantar los contenedores, los *logs* del servicio web mostraron el proceso exitoso de validación y obtención del certificado, resultando en una instancia de Jitsi plenamente funcional y accesible a través de HTTPS con un certificado de confianza.

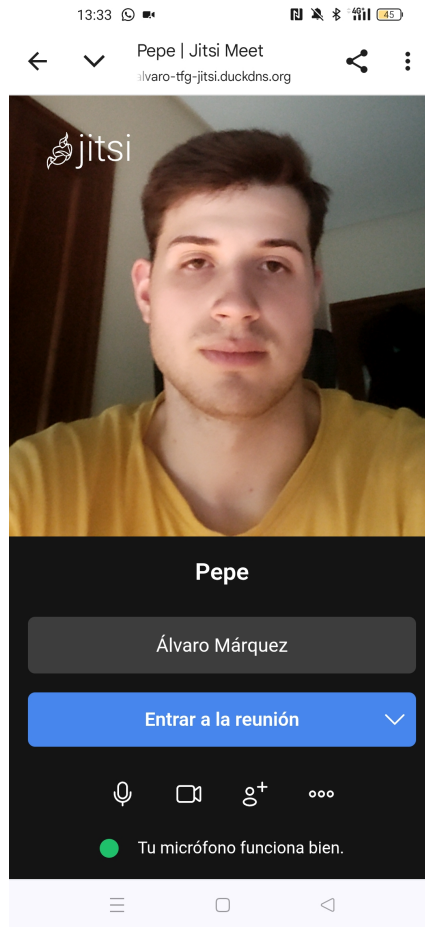


Figura 5.6: Captura de la sala de entrada a la reunión de Jitsi sin errores

Resolución de Desafíos Finales en Jibri

Con la plataforma Jitsi estable, se realizaron las primeras pruebas de grabación con Jibri, que revelaron dos problemas finales:

1. **Pantalla Negra en las Grabaciones:** El problema más crítico fue que los vídeos generados por Jibri contenían el audio correcto, pero la

imagen era una pantalla negra. Tras una investigación muy exhaustiva no se ha podido solucionar esta problemática por lo tanto se dejara como linea de trabajo futura y mejora del proyecto.

2. **Conexión Local de Jibri:** Se observó que, aunque los clientes externos podían conectarse a Jitsi sin problemas, Jibri (que se conecta desde dentro de la red Docker) fallaba al resolver la dirección del servidor. Esto finalmente paso a ser normalizado ya que la propia maquina Debian funciona como servidor que permite la conexión de los distintos dispositivos externos y no necesariamente debe funcionar localmente, al menos esa no es la intención principal, también puede dejarse pendiente como linea de trabajo futura.

Despliegue y Validación del Pipeline de Procesamiento

Con el sistema de captura de vídeo completamente funcional, el último paso fue desplegar los componentes de procesamiento y validar el flujo de datos de extremo a extremo.

- **Implementación del *Script* de Procesamiento:** Se desarrolló el *script* final en PySpark (`main.py`), incorporando la lógica de procesamiento inteligente que se describió en la fase de diseño. Este *script* es capaz de detectar los vídeos más antiguos en el directorio de entrada, procesarlos, mover los ficheros originales a una carpeta de archivado y evitar el reprocesamiento. Se resolvieron, además, conflictos de dependencias, como el de NumPy con OpenCV, fijando las versiones en un fichero `requirements.txt` que se instala durante la construcción de la imagen Docker.
- **Prueba de Concepto y Validación End-to-End:** Se realizó una prueba completa del sistema. Una sesión de vídeo fue grabada con Jitsi, y el archivo MP4 resultante fue colocado en el directorio `/data`. A continuación, se ejecutó el *pipeline* de procesamiento. El análisis de los *logs* de la aplicación demostró que el *script* localizó el vídeo, lo procesó aplicando la transformación de inversión de color, guardó el nuevo vídeo en la carpeta `/data/processed` y finalmente archivó el vídeo original. Este exitoso resultado validó la arquitectura completa y el correcto funcionamiento de todos los componentes integrados.

```

alvarodebian-tfg-alvaro:~/Documentos/GitHub/TFG-SISTEMA-PROCESAMIENTO-VIDEO/src$ docker compose up -d
[+] Running 5/5
 ✓ Network src_processing_net      Created
 ✓ Container src-spark-master-1    Started
 ✓ Container src-kafka-1          Started
 ✓ Container src-spark-worker-1    Started
 ✓ Container src-python-processor-1 Started
alvarodebian-tfg-alvaro:~/Documentos/GitHub/TFG-SISTEMA-PROCESAMIENTO-VIDEO/src$ docker compose logs -f python-processor
python-processor-1 | WARNING: Using incubator modules: jdk.incubator.vector
python-processor-1 | Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
python-processor-1 | Setting default log level to "WARN".
python-processor-1 | To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
python-processor-1 | 25/07/03 16:32:09 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
python-processor-1 | ¡Conexión con Spark establecida con éxito!
python-processor-1 | =====
python-processor-1 | Procesando el video más antiguo: /app/data/61988609-c033-4431-970c-4ee1e1881e9/testtfg_2025-06-23-13-26-00.mp4
python-processor-1 | Aplicando transformación simple...
python-processor-1 | Procesados 670 frames.
python-processor-1 | Video procesado guardado en: /app/data/processed/procesado_testtfg_2025-06-23-13-26-00.mp4
python-processor-1 | Sesión de Spark detenida.
python-processor-1 | =====
python-processor-1 | python-processor-1 exited with code 0

```

Figura 5.7: Captura donde se aprecia todos los contenedores funcionales y el script sin errores.

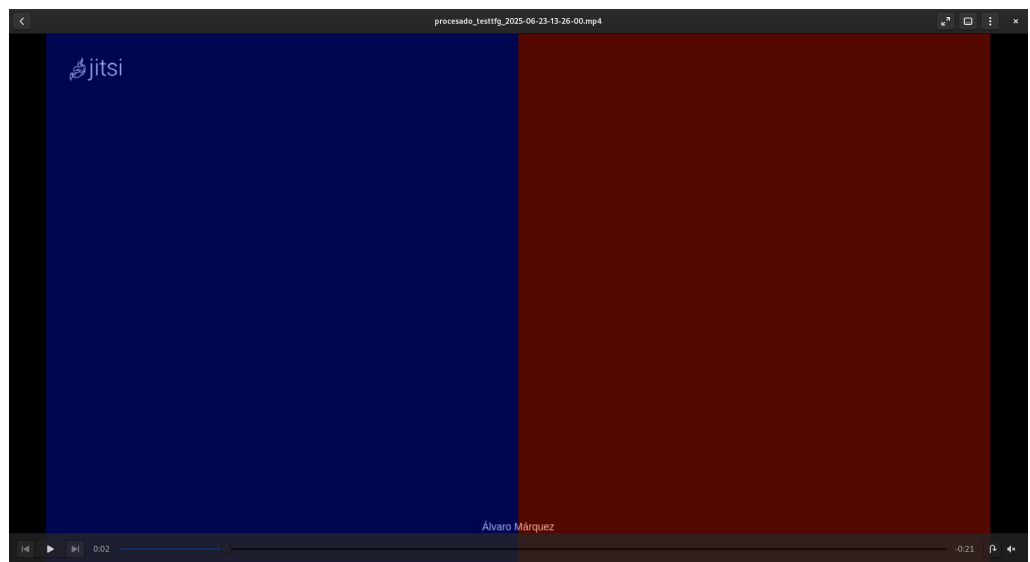


Figura 5.8: Captura donde se aprecia el video ya procesado mediante la transformación de color.

Incorporación de Recursos Visuales

Para mejorar la comprensión de la arquitectura y el flujo de trabajo descritos en este capítulo, se ha creado un conjunto de diagramas que se pueden encontrar en el Apéndice De Diseño. Estos recursos visuales incluyen un diagrama de la arquitectura general del sistema, un diagrama del pipeline

de datos y un diagrama de despliegue de los contenedores Docker, entre otros. Estos elementos son fundamentales para clarificar las complejas interacciones entre los distintos componentes del sistema.

Capítulo 6

Trabajos relacionados

Para contextualizar adecuadamente el alcance y la contribución de este TFG, es fundamental analizar los proyectos e investigaciones previas que han servido como base o que abordan problemas similares. Este capítulo revisa los trabajos más influyentes, desde el proyecto directo del que este TFG es una continuación, hasta investigaciones académicas que validan el uso de las tecnologías seleccionadas.

6.1. Proyecto de Origen: El Sistema FIS-FBIS

Este Trabajo de Fin de Grado es una continuación directa del proyecto **FIS-FBIS** [1], un sistema de bajo coste para la telerehabilitación de pacientes con Parkinson desarrollado en la Universidad de Burgos. El objetivo de FIS-FBIS era crear una plataforma completa que permitiera la comunicación entre paciente y terapeuta, la grabación de sesiones y, fundamentalmente, la evaluación automática de los ejercicios.

El sistema original fue el resultado de dos Trabajos de Fin de Máster que funcionaban de manera conjunta y complementaria: uno enfocado en la infraestructura de datos (TFM-FIS-IF) y otro en la inteligencia artificial para el análisis de vídeo (TFM-FIS-IA). Aquí proporciono el artículo científico relacionado [8].

TFM-FIS-IF: Infraestructura de Datos para Procesamiento de Vídeo

El trabajo de José Luis Garrido Labrador [9] se centró en la creación de una arquitectura Big Data basada en colas de mensajería para el procesamiento de vídeo en tiempo real. Su sistema ya utilizaba **Apache Kafka** como bus de eventos y **Apache Spark** para el procesamiento. La ingesta de datos se realizaba mediante scripts de Python (`'emitter.py'`, `'producer.py'`) que simulaban un flujo de vídeo y lo enviaban a Kafka para ser procesado por un consumidor de Spark (`'consumer.py'`).

Si bien este TFM sentó las bases de la arquitectura de este proyecto, se trataba de una prueba de concepto. Este TFG toma esta arquitectura como punto de partida y la evoluciona, reemplazando los scripts de simulación por una integración real con **Jitsi/Jibri** y profesionalizando todo el despliegue mediante **Docker y Docker Compose** para garantizar un entorno robusto, escalable y fácil de gestionar.

TFM-FIS-IA: Visión Artificial para Comparación de Posiciones

De forma paralela, el trabajo de José Miguel Ramírez Sanz [16] abordó la parte de inteligencia artificial del sistema. Su TFM se centró en el estudio del estado del arte de las herramientas de visión artificial para Python y en la implementación de un sistema de comparación de posiciones. El objetivo era extraer el esqueleto humano de los fotogramas de vídeo para poder comparar los ejercicios realizados por los pacientes con los de los terapeutas.

La sinergia entre ambos TFMs es clara: el TFM de infraestructura (IF) proporcionaba el pipeline de datos, mientras que el de inteligencia artificial (IA) aportaba la lógica de análisis. Este trabajo se centra en mejorar y modernizar la parte de infraestructura (IF) para que pueda dar un soporte más sólido a futuras aplicaciones de análisis como la desarrollada en el TFM-IA.

Arquitecturas para el Análisis de Vídeo en Tiempo Real con Spark

Más allá de los proyectos del entorno de la UBU, se ha revisado la literatura académica para validar la elección de la arquitectura. Un ejemplo es el artículo *Distributed Real-Time Video Stream Analytics on top of Spark*

[11]. En este trabajo, los autores proponen una arquitectura genérica para el análisis de flujos de vídeo en tiempo real utilizando Spark Streaming.

El estudio demuestra cómo Spark es capaz de procesar flujos de datos para realizar tareas como la detección de objetos. Su enfoque valida la elección de Spark como motor de procesamiento para este TFG, dada su capacidad para manejar la carga computacional del análisis de vídeo de forma distribuida. Sin embargo, este proyecto se diferencia y amplía este enfoque al:

- Integrar una solución completa de captura de vídeo para telemedicina como es **Jitsi/Jibri**, en lugar de fuentes de vídeo genéricas.
- Implementar **Apache Kafka** como un bus de mensajería intermedio, lo que proporciona un mayor desacoplamiento y tolerancia a fallos entre la captura y el procesamiento.
- Enfocar la arquitectura en un caso de uso específico (la telerehabilitación), en lugar de en el análisis genérico de vídeo.

Capítulo 7

Conclusiones y Líneas de trabajo futuras

Para finalizar esta memoria, en este capítulo se presentan, por una parte, las conclusiones principales que se han podido extraer del desarrollo del proyecto y, por otra, las posibles líneas de trabajo futuras que se abren a partir de la infraestructura implementada.

7.1. Conclusiones

La realización de este TFG ha permitido obtener una serie de conclusiones tanto a nivel técnico como a nivel de gestión y desarrollo de un proyecto de ingeniería de software.

Como primera conclusión, y una de las más importantes, se ha podido constatar que la **elección del entorno de desarrollo es crítica** en proyectos que involucran múltiples servicios de red y sistemas de ficheros, como es este caso. El intento inicial de despliegue sobre Windows con WSL2, aunque teóricamente viable, demostró ser una fuente de problemas de compatibilidad (especialmente de permisos y redes) muy difíciles de diagnosticar y resolver. El pivote estratégico a un entorno Linux nativo (Debian 12) fue una decisión fundamental que no solo solucionó los problemas, sino que validó la importancia de trabajar sobre un sistema operativo estándar en entornos de servidor para garantizar la estabilidad y la reproducibilidad.

En segundo lugar, se ha logrado el objetivo principal de diseñar e implementar una **arquitectura de datos robusta y escalable**, sentando las bases para un sistema de telerehabilitación mucho más potente que el del

CAPÍTULO 7. CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURAS

proyecto original. La integración de Jitsi, Docker, Kafka y Spark conforma un pipeline de datos modular y desacoplado, que es el estándar de facto en la industria para este tipo de aplicaciones. Aunque la implementación final se ha centrado en un flujo *offline*, la arquitectura está preparada para evolucionar hacia el procesamiento en tiempo real.

Finalmente, una gran parte del trabajo de ingeniería de este TFG no ha sido la programación de algoritmos complejos, sino la **integración, configuración y depuración** de múltiples tecnologías de código abierto. Enfrentarme a *logs* de errores, diagnosticar problemas de red con herramientas como *openssl*, y comprender las complejas interacciones entre los distintos contenedores ha sido un aprendizaje práctico inmenso y una parte tan valiosa como el propio desarrollo de código.

Conclusiones Personales

A nivel personal, este Trabajo de Fin de Grado ha supuesto un desafío considerable y, a su vez, una experiencia de aprendizaje inmensamente valiosa. Me ha permitido aplicar los conocimientos teóricos del Grado y los obtenidos por otros medios a un problema real y complejo, enfrentándome a las dificultades que surgen fuera del entorno académico. La necesidad de investigar, diagnosticar errores y tomar decisiones estratégicas para reconducir el proyecto me ha proporcionado una visión práctica de la ingeniería de software que considero fundamental para mi futuro profesional.

7.2. Líneas de trabajo futuras

El sistema implementado en este TFG es una base sólida sobre la que se pueden construir numerosas mejoras y nuevas funcionalidades. A continuación, se detallan algunas de las líneas de trabajo futuras más interesantes:

1. **Implementar el *Pipeline de Streaming* en Tiempo Real:** El siguiente paso natural sería evolucionar del procesamiento por lotes (*batch*) al procesamiento en tiempo real. Esto implicaría configurar Jibri para que emita un flujo de vídeo usando el protocolo RTMP, e implementar un consumidor en Spark *Streaming* que procese los fotogramas a medida que llegan a través de Kafka.
2. **Integración con los Algoritmos de Análisis de Movimiento:** Conectar este *pipeline* de datos con los algoritmos de análisis de esqueletos y DTW desarrollados en los trabajos previos (TFM-FIS-IA y

el TFG de Lucía Núñez Calvo). Esto permitiría crear un sistema completo que capture, procese y evalúe los ejercicios de forma totalmente automatizada.

3. **Escalabilidad y Orquestación Avanzada:** Aunque Docker Compose es ideal para el desarrollo, para un entorno de producción se podría migrar la arquitectura a un orquestador de contenedores más avanzado como **Kubernetes**. Esto permitiría un escalado automático de los componentes (por ejemplo, añadir más workers de Spark o más instancias de Jibri si hay muchas sesiones concurrentes).
4. **Monitorización del Sistema:** Desplegar una pila de monitorización (por ejemplo, con Prometheus y Grafana) para obtener métricas en tiempo real del estado de los brokers de Kafka, los trabajos de Spark y el resto de los servicios, asegurando la salud y el rendimiento del sistema.
5. **Mejora de la Captura de Vídeo y Audio:** Investigar y optimizar la calidad de la fuente de datos. Esto podría incluir pruebas con diferentes configuraciones de *códecs* en Jibri, el uso de cámaras de mayor resolución, o la exploración de configuraciones de audio avanzadas para garantizar que la materia prima para el análisis sea de la máxima calidad posible.

Bibliografía

- [1] ADMIRABLE Research Group. Fis-fbis: Repository of the fis project pi19/00670. <https://github.com/admirable-ubu/FIS-FBIS>, 2020. Accedido el: 06-Jul-2025.
- [2] Apache Software Foundation. Apache kafka documentation. <https://kafka.apache.org/documentation/>, 2025. Accedido el: 30-Abr-2025.
- [3] Apache Software Foundation. Apache spark™ - unified analytics engine for big data. <https://spark.apache.org/>, 2025. Accedido el: 30-Abr-2025.
- [4] Atlassian. Jira software, 2025. Disponible en: <https://www.atlassian.com/software/jira>.
- [5] Kenneth P Birman. *A guide to reliable distributed systems: building high-assurance applications and services*. Springer Science & Business Media, 2007.
- [6] Michelle A Cottrell, Shaun P O’Leary, and Trevor G Russell. Telerehabilitation for people with parkinson’s disease. *Journal of telemedicine and telecare*, 23(6):594–603, 2017.
- [7] Docker, Inc. Docker documentation, 2025. Disponible en: <https://docs.docker.com/>.
- [8] José L. Garrido-Labrador, José M. Ramírez-Sanz, Jonathan Latorre-Jaén, José F. Díez-Pastor, Diego López-de Ipiña, and Esther Cubo. Fis-hub: A low-cost tele-rehabilitation system for patients that suffer from parkinson disease. *Healthcare*, 11(4):507, 2023.

- [9] José Luis Garrido Labrador. Tfm-fis-if: Big data architecture of queues for real time video processing. <https://github.com/jlgarridol/TFM-FIS-IF>, 2020. Accedido el: 19-Jun-2025.
- [10] Jitsi Community and 8x8, Inc. Jitsi handbook documentation. <https://jitsi.github.io/handbook/docs/intro>, 2025. Accedido el: 30-Abr-2025.
- [11] Karim Karimov, Tatyana Tsymbol, Sergey Konyagin, Andrey Kustov, Elena Tutubalina, and Pavel Braslavski. Distributed real-time video stream analytics on top of spark. In *Proceedings of the 24th conference on computational linguistics and intellectual technologies (Dialogue 2018)*, 2018.
- [12] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, Inc., 2017.
- [13] OpenSSL Software Foundation. Openssl command-line howto: s_client. https://www.openssl.org/docs/man3.0/man1/openssl-s_client.html, 2024. Accedido el: 06-Jul-2025.
- [14] Tamara Pringsheim, Nathalie Jette, anatoliy Frolkis, and T. D. L. Steeves. The prevalence of parkinson's disease: a systematic review and meta-analysis. *Movement disorders*, 29(13):1583–1590, 2014.
- [15] Python Software Foundation. Python language reference, 2025. Disponible en: <https://www.python.org>.
- [16] José Miguel Ramírez Sanz. Tfm-fis-ia: Estudio del estado del arte de las herramientas de visión artificial para python e implementación de un sistema de comparación de posiciones. <https://github.com/Josemi/TFM-FIS-IA>, 2020. Accedido el: 19-Jun-2025.
- [17] Eric Rescorla. *SSL and TLS: designing and building secure systems*. Addison-Wesley Professional, 2001.
- [18] The Jitsi Team. docker-jitsi-meet: Jitsi meet on docker. <https://github.com/jitsi/docker-jitsi-meet>, 2024. Accedido el: 06-Jul-2025.
- [19] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium*

on Networked Systems Design and Implementation (NSDI 12), pages
15–28, 2012.