

Grado en Ingeniería Informática
Curso 2023-2024

Trabajo Fin de Grado

“Análisis, diseño e implementación de un sistema de ficheros básico basado en MQTT”

Diego Fuentes Antúnez

Tutor

Alejandro Calderón Mateos

Leganés, septiembre de 2024



Esta obra se encuentra sujeta a la licencia Creative Commons
Reconocimiento – No Comercial – Sin Obra Derivada

RESUMEN

MQTT (Message Queuing Telemetry Transport) es un protocolo de comunicación ampliamente utilizado en el ámbito del Internet de las Cosas, principalmente por su eficiencia en redes con un ancho de banda limitado. Se basa en una arquitectura de publicación y suscripción.

Aunque MQTT suele asociarse principalmente con el uso de sensores y actuadores, se ha decidido explorar en el presente trabajo una vía alternativa de aplicación, mediante su uso en un sistema de ficheros. El propósito es capturar los beneficios del uso tradicional del mismo y extrapolarlos a esta nueva área. De este modo el proyecto puede concebirse como el desarrollo de un sistema de ficheros distribuido o de red, pero con la innovación que supone implementarlo con un protocolo de comunicación MQTT.

Además, se van a crear distintos clientes, y cada uno de ellos va a tener una funcionalidad asociada, siguiendo un enfoque de diseño modular. Por ejemplo, se va a desarrollar un cliente que registre todos los eventos dentro del sistema, y posteriormente se va a conectar con una herramienta de monitorización, concretamente a la pila ELK (Elasticsearch, Logstash, Kibana). Todo este proceso sirve para realzar la versatilidad que ofrece la arquitectura pub/sub con la integración de nuevos módulos.

Para que todo esto sea posible y el usuario pueda interactuar con los ficheros que se encuentran en el almacenamiento dentro del nodo externo, es necesario que el usuario pueda interactuar con el sistema a través de una interfaz. Para ello se ha decidido hacer uso de FUSE (Filesystem in Userspace), que permite crear un sistema de ficheros en el espacio de usuario sin necesidad de modificar el kernel.

En la presente memoria se va a documentar todo el proceso que se ha seguido para la realización del proyecto. Este proceso sigue las etapas de Análisis, Diseño, Implementación, Implantación, Evaluación y Planificación.

Palabras clave: Sistema de ficheros distribuido, Protocolo de comunicación, Sistema de archivos en el espacio de usuario, logs, monitorización.

AGRADECIMIENTOS

Quiero agradecer este trabajo a las dos personas más importantes de mi vida.

A mi novia Noelia por el apoyo incondicional que me ha dado estos últimos años, y por haber sido un pilar imprescindible en la finalización de mis estudios.

A mi padre, que me enseñó a ser resiliente y a pelear en la adversidad. Estos valores han estado presentes durante esta etapa de mi vida aunque no hayas podido formar parte de ella, porque tu recuerdo siempre me acompaña.

ÍNDICE DE CONTENIDOS

1.	Introducción	1
1.1	Motivación.....	1
1.2	Objetivos.....	1
1.3	Estructura del documento	1
2.	Estado de la cuestión	4
2.1	Conceptos relevantes para el proyecto	4
2.1.1	Sistema de ficheros	4
2.1.2	Sistema de ficheros distribuido/Sistema de ficheros de red	4
2.1.3	Protocolo de comunicación	4
2.1.4	Callbacks	5
2.1.5	Espacio de usuario vs Espacio del kernel.....	5
2.1.6	Sistema de ficheros virtual	5
2.2	Soluciones similares	5
2.2.1	Network File System	6
2.2.2	LBFS (Low-Bandwidth File System).....	6
2.2.3	Andrew File System	8
2.2.4	Coda.....	8
2.2.5	Tabla comparativa	10
2.3	Tecnologías.....	10
2.3.1	MQTT.....	10
2.3.2	FUSE (Filesystem in Userspace).....	12
2.3.3	ELK: Elasticsearch, Logstash y Kibana	14
2.3.4	Entorno de desarrollo.....	15
2.3.4.1	Linux Ubuntu 22.04.....	15
2.3.4.2	Python.....	15
2.3.5	Bibliotecas utilizadas	16
2.4	Marco regulador	17
2.4.1	Marco legislativo	17
2.4.2	Estándares técnicos.....	18
2.4.3	Uso de licencias	18
2.5	Impacto socioeconómico	19
2.5.1	Impacto social.....	19
2.5.2	Impacto económico.....	19

2.5.3	Impacto medioambiental	20
3.	Análisis, Diseño, Implementación e Implantación.....	21
3.1	Metodología.....	21
3.2	Análisis	23
3.2.1	Descripción detallada de la solución	23
3.2.2	Requisitos	24
3.2.2.1	Requisitos funcionales	26
3.2.2.2	Requisitos no funcionales.....	33
3.2.3	Casos de uso	35
3.2.4	Matriz de trazabilidad.....	39
3.3	Diseño.....	40
3.3.1	Arquitectura del sistema	40
3.3.2	Diagrama de clases	43
3.3.2.1	Cardinalidad	46
3.3.2.2	Funciones FUSE.....	46
3.3.3	Diagramas de secuencia.....	47
3.3.3.1	Diagrama de secuencia de operación con respuesta.....	47
3.3.3.2	Diagrama de secuencia de operación sin respuesta	49
3.3.3.3	Diagrama de secuencia del sistema de logging	51
3.4	Implementación	52
3.4.1	Estructura del proyecto.....	52
3.4.2	Repaso de las distintas funciones existentes dentro del código.....	54
3.4.2.1	fuse_client.py.....	54
3.4.2.2	file_manager.py	55
3.4.3	Dificultades encontradas en la etapa de implementación.....	55
3.5	Implantación	56
3.5.1	Instalación de dependencias	56
3.5.2	Arranque y configuración.....	57
3.5.3	Configuración con Mosquitto.....	57
3.5.4	Despliegue del sistema	59
3.5.5	Implantación de ELK.....	59
4.	Evaluación.....	61
4.1	Diseño del plan de pruebas.....	61
4.2	Casos de prueba	62
5.	Visualización con Kibana y procesamiento con Logstash	68

5.1	Archivo de configuración de Logstash.....	68
5.2	Visualización de logs y dashboards.....	68
6.	Planificación y presupuesto.....	79
6.1	Planificación	79
6.2	Presupuesto.....	82
6.2.1	Costes de Hardware	82
6.2.2	Costes de Software	82
6.2.3	Costes de personal	83
6.2.4	Costes indirectos.....	83
6.2.5	Costes totales	83
7.	Conclusiones y trabajos futuros	85
7.1	Objetivos cumplidos	85
7.2	Conclusiones personales.....	85
7.3	Trabajos futuros.....	86
	Bibliografía.....	89
	Anexo. Summary.....	92

ÍNDICE DE FIGURAS

Figura 1. Arquitectura de NFS [6].....	6
Figura 2. Problema que resuelve LBFS [7].....	7
Figura 3. Arquitectura de LBFS [7]	7
Figura 4. Arquitectura de AFS [8].....	8
Figura 5. Arquitectura de FUSE [11]	13
Figura 6. Flujo de datos de ELK [14].....	15
Figura 7. Ciclo de vida en cascada [27]	21
Figura 8. Diagrama de casos de uso	36
Figura 9. Arquitectura de la solución	41
Figura 10. Diagrama de clases	45
Figura 11. Diagrama de secuencia I	48
Figura 12. Diagrama de secuencia II.....	50
Figura 13. Diagrama de secuencia III	51
Figura 14. Función sync	54
Figura 15. Creación del cliente MQTT con userdata	54
Figura 16. Callback on_message del cliente FUSE.....	55
Figura 17. Registros log	55
Figura 18. Implementación de la función de escritura	56
Figura 19. Pantalla Discover de Kibana	69
Figura 20. Expansión de documentos en Kibana	70
Figura 21. Filtrado de documentos en Kibana.....	71
Figura 22. Mensajes log de MQTT	72
Figura 23. Mensajes de log de FUSE	73
Figura 24. Mensajes de error en Kibana.....	74
Figura 25. Creación de un gráfico en Kibana.....	76
Figura 26. Dashboard en Kibana.....	77
Figura 27. Diagrama de Gantt	81
Figura 28. Tiempo de arranque I	87
Figura 29. Tiempo de arranque II.....	87

ÍNDICE DE TABLAS

Tabla 1. Tabla comparativa de soluciones alternativas	10
Tabla 2. Definición de los callbacks de MQTT.....	11
Tabla 3. Explicación QoS de MQTT.....	11
Tabla 4. Estudio de alternativas de lenguajes de programación.....	16
Tabla 5. Bibliotecas utilizadas en el proyecto	17
Tabla 6. Plantilla de requisitos	25
Tabla 7. RF-01	26
Tabla 8. RF-02	26
Tabla 9. RF-03.....	26
Tabla 10. RF-04.....	26
Tabla 11. RF-05	27
Tabla 12. RF-06.....	27
Tabla 13. RF-07	27
Tabla 14. RF-08	27
Tabla 15. RF-09.....	28
Tabla 16. RF-10.....	28
Tabla 17. RF-11	28
Tabla 18. RF-12.....	28
Tabla 19. RF-13	29
Tabla 20. RF-14.....	29
Tabla 21. RF-15	29
Tabla 22. RF-16.....	29
Tabla 23. RF-17.....	30
Tabla 24. RF-18.....	30
Tabla 25. RF-19.....	30
Tabla 26. RF-20.....	31
Tabla 27. RF-21	31
Tabla 28. RF-22	31
Tabla 29. RF-23	31
Tabla 30. RF-24.....	32
Tabla 31. RF-25.....	32
Tabla 32. RF-26.....	32
Tabla 33. RF-27	33
Tabla 34. RNF-01	33
Tabla 35. RNF-02	33
Tabla 36. RNF-03	33
Tabla 37. RNF-04	34
Tabla 38. RNF-05	34
Tabla 39. RNF-06.....	34
Tabla 40. RNF-07	34
Tabla 41. RNF-08.....	35

Tabla 42. RNF-09	35
Tabla 43. RNF-10	35
Tabla 44. RNF-11	35
Tabla 45. Plantilla de casos de uso	36
Tabla 46. CU-01	37
Tabla 47. CU-02	38
Tabla 48. CU-03	38
Tabla 49. CU-04	38
Tabla 50. CU-05	39
Tabla 51. Matriz de trazabilidad I.....	40
Tabla 52. Funciones de fusepy	47
Tabla 53. Plantilla de casos de prueba.....	61
Tabla 54. P-01.....	62
Tabla 55. P-02.....	62
Tabla 56. P-03.....	62
Tabla 57. P-04.....	63
Tabla 58. P-05.....	63
Tabla 59. P-06.....	63
Tabla 60. P-07.....	63
Tabla 61. P-08.....	64
Tabla 62. P-09.....	64
Tabla 63. P-10.....	64
Tabla 64. P-11.....	64
Tabla 65. P-12.....	65
Tabla 66. P-13.....	65
Tabla 67. P-14.....	65
Tabla 68. P-15.....	65
Tabla 69. P-16.....	66
Tabla 70. P-17.....	66
Tabla 71. P-18.....	66
Tabla 72. P-19.....	66
Tabla 73. P-20.....	67
Tabla 74. P-22.....	67
Tabla 75. P-22.....	67
Tabla 76. Planificación del proyecto	79
Tabla 77. Costes de equipo	82
Tabla 78. Costes de licencias de Software	82
Tabla 79. Costes de personal	83
Tabla 80. Costes indirectos.....	83
Tabla 81. Costes totales	83
Tabla 82. Presupuesto con beneficio	84
Tabla 83. Posibles implementaciones futuras.....	88

1. INTRODUCCIÓN

1.1 Motivación

A diario interactuamos con sistemas de ficheros que permiten la organización de todos aquellos documentos que forman parte de nuestro día a día, como manejar archivos PDF que solemos destinar a tareas de trabajo, fotos para recordar momentos de nuestra vida, etc. Tal vez utilices el sistema de ficheros de tu sistema operativo, o alguna plataforma de almacenamiento, como Google Drive. El problema de estos sistemas de ficheros es que son muy limitados en cuanto a la configuración básica que ofrecen, aunque existan pequeñas variaciones entre las mismas, y su creación requiere de unos conocimientos técnicos que pueden no estar al alcance de todos.

El objetivo de este trabajo es ofrecer un sistema de ficheros de red al usuario final, que cuente con las siguientes características:

- Eficiente en redes con un ancho de banda limitado.
- Que ofrezca una resistencia frente a la pérdida de datos.
- Que ofrezca configurabilidad y personalización en función del cliente que acceda al sistema.
- Opción de poder monitorizar las operaciones del sistema de ficheros, y poder ofrecer estadísticas que sean de utilidad para el usuario.

También se busca la transparencia de red, dando la ilusión de que se está accediendo a los ficheros de forma local, y que la interfaz que se ofrezca para la manipulación del sistema esté integrada con el sistema operativo. Por último, se busca que la aplicación sea fácil de usar desde la perspectiva del usuario.

1.2 Objetivos

Antes de empezar a realizar el trabajo, se han establecido los siguientes 4 objetivos:

1. Implementar un sistema de ficheros básico completamente funcional.
2. Proporcionar una interfaz amigable para poder interactuar con el sistema.
3. Establecer una conexión entre los clientes MQTT para la correcta gestión de la información de los ficheros.
4. Establecer funcionalidades adicionales no convencionales al uso de un sistema de ficheros, mediante el uso del protocolo de publicación-suscripción y diversos clientes MQTT. En este caso se quiere implementar un logger que permita la monitorización del sistema y conectarlo con la pila ELK.

1.3 Estructura del documento

En este apartado se va a explicar la estructura que sigue el documento, el cual se divide en los siguientes capítulos:

- **Introducción**

Primero se explica la motivación detrás de la elección del tema que se va a tratar, exponiendo y justificando las causas. Después de la motivación se exponen los objetivos que se han de cumplir con la realización de este trabajo. Para finalizar se describe el contenido del documento.

- **Estado de la cuestión**

El estado de la cuestión es un análisis de la relevancia y originalidad del proyecto, que abarca los siguientes aspectos:

- **Estudio de conceptos**

Se definen los conceptos asociados al desarrollo del proyecto, los cuales van a ser mencionados en repetidas ocasiones, y que es necesario explicar para proporcionar el correspondiente contexto.

- **Estudio de soluciones similares**

Para estudiar la originalidad del proyecto y justificar la innovación que aporta, es necesario comparar la solución aquí presentada con las distintas aportaciones de otros investigadores dentro del mismo campo de estudio.

- **Estudio de tecnologías relevantes para el desarrollo de la solución**

Se va a realizar un estudio de las tecnologías que se pueden usar para poder realizar el desarrollo de la solución, justificando su elección y explicando los conceptos técnicos relevantes.

- **Marco regulador**

Se estudia la viabilidad del proyecto dentro del marco legal, y las licencias propias de las tecnologías usadas en el proyecto.

- **Impacto socioeconómico**

Se hace un estudio del impacto que tiene el proyecto en la sociedad, en el ámbito económico y en el ámbito medioambiental.

- **Análisis, diseño, implementación e implantación**

En este capítulo se documenta todo el proceso típico de un proyecto de software. Abarca las siguientes etapas:

- **Metodología**

Se describe el ciclo de vida que ha servido a modo de guía para organizar todas las fases del proceso.

- **Análisis**

El análisis, al tratarse de la primera etapa del proceso, compone todas aquellas actividades destinadas a especificar todos los posibles aspectos

del proyecto para tener una base sobre la cuál trabajar. Primero se define de una forma genérica y aproximada como se desea desarrollar la solución, después se produce la elicitación de requisitos y diagramas de casos de uso, y por último se estudia la relación entre ambos mediante una matriz de trazabilidad.

- **Diseño**

A partir de las tareas realizadas en la etapa de análisis, se conceptualiza la solución a través de los siguientes diagramas:

- Diagrama de arquitectura
- Diagrama de clases
- Diagramas de secuencia

- **Implementación**

Durante esta fase se desarrolla y codifica la solución. En este capítulo se va a explicar la estructura del directorio del proyecto, el propósito de todas las funciones de cada fichero Python, y los aspectos que más dificultad han supuesto durante la programación.

- **Implantación**

Se explican detalladamente los pasos a seguir para poder desplegar la solución. Esto comprende la instalación de bibliotecas y dependencias, las variables de configuración que es necesario codificar, y la ejecución de scripts.

- **Evaluación**

Esta fase es primordial para asegurar el correcto funcionamiento y calidad del software desarrollado. Se diseña un plan de pruebas, y para cada prueba se verifica su cumplimiento.

- **Planificación y presupuesto**

Para la planificación se tiene en consideración el ciclo de vida elegido en el epígrafe de **Metodología**. De este proceso, se obtiene el total de horas que conlleva la realización del proyecto en su totalidad, y se utiliza dicho dato para realizar el presupuesto, calculando el coste del salario total y los tiempos de depreciación del equipo utilizado, entre otros aspectos.

- **Conclusiones y trabajos futuros**

En este capítulo se exponen unas conclusiones donde se realiza una reflexión sobre el cumplimiento de los objetivos, las dificultades que ha conllevado el proceso, el aprendizaje que se ha producido, y también se relacionan las habilidades necesarias para realizar un trabajo de esta índole con las habilidades adquiridas durante el grado.

Por otro lado, se profundiza en las posibles mejoras que se pueden introducir en el desarrollo del sistema para poder completar la solución.

2. ESTADO DE LA CUESTIÓN

2.1 Conceptos relevantes para el proyecto

Durante este apartado se van a explicar conceptos relevantes que se van a ir presentando en este documento.

2.1.1 Sistema de ficheros

Un sistema de ficheros es aquel conjunto de servicios que permite la manipulación y almacenamiento de los datos, y generalmente es proporcionado a través de un sistema operativo. Las estructuras donde se almacenan esos datos suelen llamarse ficheros o archivos, y las estructuras que almacenan un conjunto de archivos se denominan directorios o carpetas. Un directorio está considerado como un tipo especial de fichero, pero se van a utilizar ambos conceptos como términos independientes a lo largo de este documento por comodidad.

Su manipulación se suele proporcionar al usuario a través de una interfaz.

2.1.2 Sistema de ficheros distribuido/Sistema de ficheros de red

Un sistema de ficheros distribuido o *Distributed File System* (DFS, por sus siglas en inglés), es un sistema de ficheros que se basa en una arquitectura cliente-servidor. El almacenamiento puede estar distribuido entre uno o más servidores, y se permite la conexión de más de un cliente.

Características habituales de este tipo de sistemas son la tolerancia a fallos y la replicación. Estas características se van a implementar dentro de la solución.

A veces se nombra al mismo concepto como “Sistema de ficheros de red”, aunque este concepto se suele asociar únicamente al protocolo NFS, que se verá más adelante. Del mismo modo, se suele utilizar el término DFS solo para aquellos sistemas que cuentan con múltiples nodos para el almacenamiento de los datos. Se ha podido comprobar en la investigación de este tema que existe un poco de ambigüedad entre ambos términos.

Como se pretende implementar un nodo que replique los datos de los archivos (aunque no en el sentido tradicional en el que estos sistemas suelen operar), se ha decidido usar esta terminología por comodidad para describir el sistema que se pretende desarrollar.

2.1.3 Protocolo de comunicación

Un protocolo de comunicación describe un conjunto de reglas y descripciones formales que deben seguir los mensajes digitales que se intercambian entre sistemas.

[1]

Esto proporciona un estándar que facilita la comunicación. Para este proyecto se va a utilizar MQTT, que se va a explicar más adelante.

2.1.4 Callbacks

Una función callback es una función que se pasa como argumento a otra función y es ejecutada dentro de esta en un momento determinado. La idea es modularizar el código para que se pueda abstraer parte de la lógica y ejecutarla independientemente de la función que llama al callback. [2]

MQTT utiliza un paradigma de programación asíncrona, y se hace uso de callbacks en respuesta a los distintos eventos que se producen durante la ejecución del programa (on_connect, on_message, etc.).

2.1.5 Espacio de usuario vs Espacio del kernel

Para explicar el funcionamiento de la solución se van a mencionar estos términos ocasionalmente.

El espacio de usuario es el espacio de memoria donde se ejecutan las aplicaciones que no pueden acceder a los recursos del sistema, y que deben realizar llamadas al kernel para que se le proporcionen dichos recursos, mientras que el espacio del kernel es el responsable de proporcionar dichos recursos del sistema, mediante el uso de una interfaz. [3]

2.1.6 Sistema de ficheros virtual

El sistema de ficheros virtual (VFS, o Virtual File System), es una capa de software en el kernel del Sistema Operativo que proporciona a los programas en el espacio de usuario la interfaz necesaria para poder interactuar con un sistema de ficheros. También proporciona una capa de abstracción dentro del kernel que permite que varias implementaciones de sistemas de ficheros coexistan. [4]

Gracias a esta parte integral de Linux, va a ser posible la implementación del sistema de ficheros personalizado, puesto que la capa de abstracción es esencial para poder integrarlo dentro del sistema operativo. Todos los sistemas de ficheros interactúan con esta parte del kernel.

2.2 Soluciones similares

A continuación, se va a hacer un estudio de las soluciones alternativas que se han encontrado, para plantearse si es realmente necesario la implementación de una solución al problema planteado, y poder estudiar qué puede aportar la solución proyectada que no se encuentre ya disponible. Se busca una solución donde:

- El sistema de ficheros sea distribuido.
- Requiera poco ancho de banda.
- Sea resistente a la pérdida de datos.
- Haga un uso transparente del protocolo de red, es decir, que parezca que se está accediendo a los ficheros de forma local.

- Ofrezca un sistema de monitorización.

2.2.1 Network File System

NFS [5], es un sistema de ficheros que permite el intercambio de archivos de forma remota para que sean accesibles por otros usuarios. NFS usa RPCs (Remote Procedure Calls) para gestionar las solicitudes entre cliente y servidor, y se caracteriza porque es:

- Transparente, haciendo parecer que se está accediendo a los ficheros de forma local
- Popular, al ser uno de los protocolos de red más usados
- Maduro
- Fácil de usar
- Multiplataforma

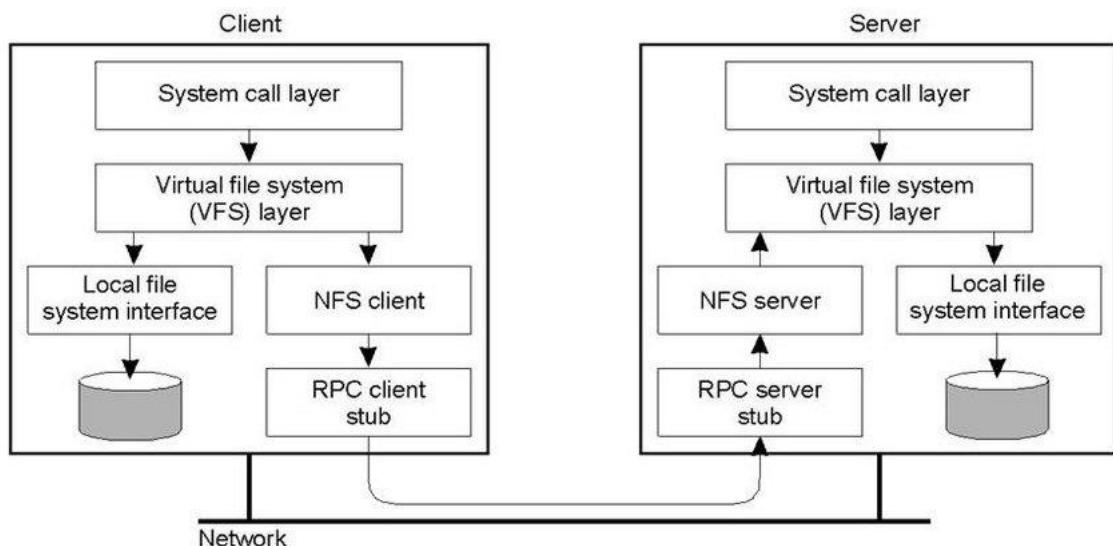


Figura 1. Arquitectura de NFS [6].

Se ha estudiado esta opción porque cumple algunos de los requerimientos que se necesitan, pero como se apreciará más adelante en la tabla comparativa, no llega a ser una solución óptima.

2.2.2 LBFS (Low-Bandwidth File System)

LBFS (Low-Bandwidth File System) [7], es un trabajo realizado por investigadores del MIT (Massachusetts Institute of technology), cuyo objetivo es proporcionar un sistema de ficheros distribuido, y que no requiera de un ancho de banda elevado. Para ello, LBFS identifica qué partes de un fichero se encuentran ya almacenadas en el lado cliente. Para hacer esto, se divide el fichero en fragmentos, que se indexan por

su valor hash utilizando el algoritmo criptográfico SHA-1. Cuando se detecta que el fragmento se repite, no se envía.

El principal problema al que se enfrentaron a la hora de diseñar este sistema de ficheros, es el de cómo reconocer los fragmentos cuando se produce un desplazamiento en la posición donde se realiza una escritura, si estos se dividen en tamaños fijos. Esto se ilustra bien con la siguiente figura (las regiones de color gris son regiones donde se ha modificado el fichero) [7].

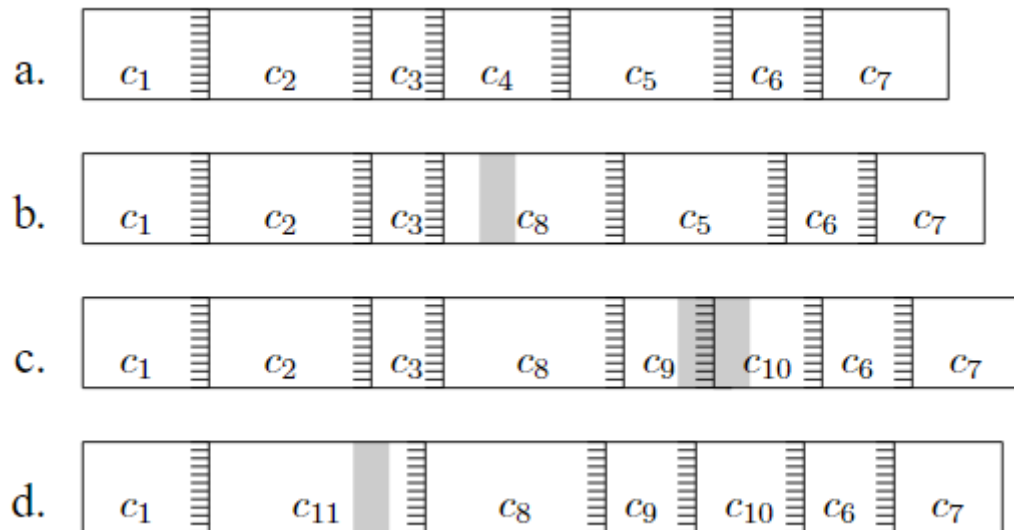


Figura 2. Problema que resuelve LBFS [7].

Para solucionar esto [7], LBFS mantiene fragmentos de datos que no se solapan y de tamaño variable, que con la implementación de un algoritmo de “ventana deslizante”, permite la identificación de los fragmentos del fichero independientemente de que estos se vayan desplazando en el mismo, examinando cada región de 48 bits, y si coincide con un determinado conjunto de bits (elegido mediante un método llamado “Rabin fingerprint”), se determina el final de un fragmento. Está basado en NFS.

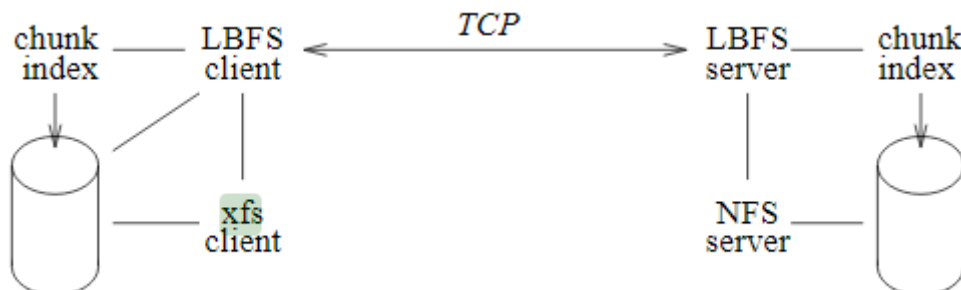


Figura 3. Arquitectura de LBFS [7]

2.2.3 Andrew File System

AFS [8] es un sistema de ficheros distribuido, que se caracteriza por el uso de una caché local en el lado cliente que minimiza la comunicación entre el cliente y el servidor, mejorando así el rendimiento y la velocidad. Esta característica hace que sea una opción interesante para el uso de redes con un bajo ancho de banda.

- Los servidores AFS (llamados “Vice”) se encargan de la gestión del sistema de ficheros y responden a las peticiones de los clientes (Venus).
- Los clientes (Venus) se encargan de interceptar las solicitudes de gestión de archivos, comprueba si el archivo está en la caché local, y si no lo está lo solicita a los servidores.

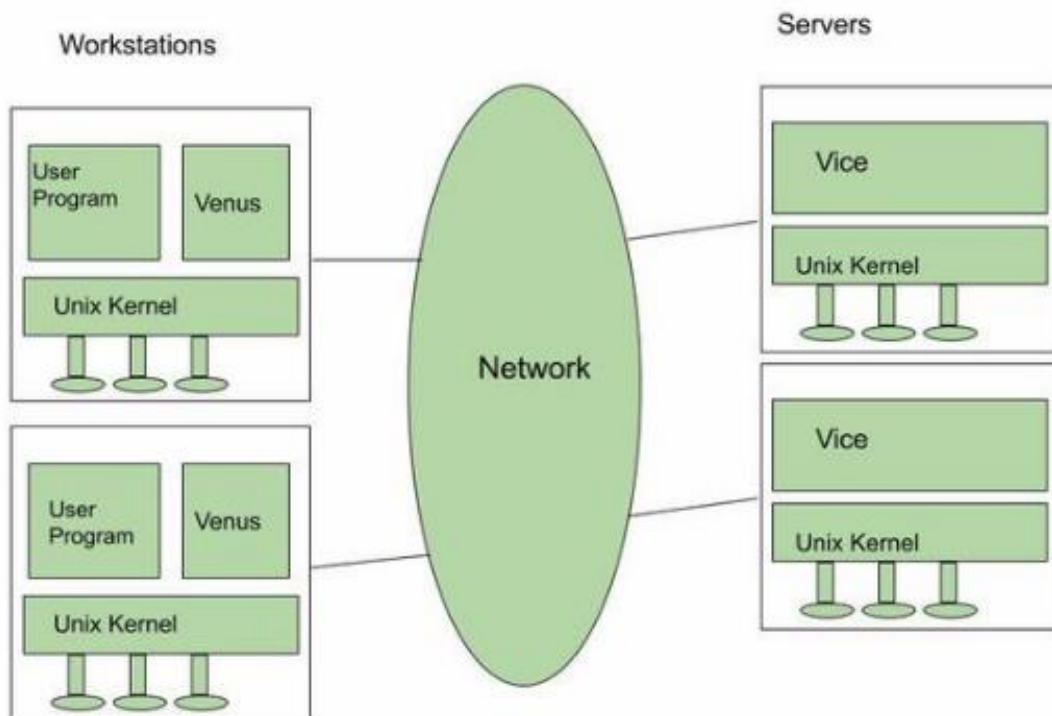


Figura 4. Arquitectura de AFS [8]

Por otro lado, también proporciona tolerancia a fallos mediante el uso de replicación, pero únicamente para volúmenes de solo lectura [9]. Esto evita conflictos de escritura y problemas de consistencia.

2.2.4 Coda

Tal y como se menciona en el artículo original [10], Coda es un sistema de ficheros distribuido inspirado en AFS, cuyo propósito es mejorar el mismo mediante la implementación de técnicas de tolerancia a fallos. Coda se inspira en AFS mediante el uso de una caché local, pero tiene una implementación más completa al introducir dos aspectos clave:

- **Replicación de volúmenes de lectura/escritura**

A diferencia de AFS, Coda utiliza una replicación de volúmenes de lectura/escritura, que se replican en múltiples servidores de forma automática. Coda sigue una estrategia optimista para poder garantizar una mayor disponibilidad.

Cuando el cliente no se puede conectar a ningún servidor, opera desconectado de la red, dependiendo únicamente de la caché. Cuando esto sucede, Coda intenta restablecer la conexión a la red tan pronto como sea posible, actualizando las réplicas presentes en todos los servidores.

- **Permite al usuario trabajar estando desconectado de la red**

Cuando el cliente trabaja desconectado de la red debido a la caída de todos los servidores, puede priorizar la lista de ficheros y directorios que se debe mantener en caché, para que pueda trabajar con éxito en caso de desconexión. Esto es importante, porque si el cacheo se realiza de forma automática, puede ser que se necesiten archivos que Coda no haya decidido cachear, en cuyo caso no sería posible trabajar desconectado de la red.

Si se producen conflictos de versiones entre ficheros, se crean repositorios temporales en cada volumen de cada réplica para que el usuario los resuelva manualmente.

Es por todos estos motivos, que Coda es la solución que más se aproxima al objetivo que se busca en el presente trabajo, ya que nos ofrece una alta tolerancia a fallos en redes con un ancho de banda limitado, replicación, y el uso de una red transparente. No obstante, no cumple con todos los requisitos que se han propuesto, y es por esta razón que se ha llevado a la implementación de una solución alternativa. A continuación, voy a exponer en una tabla comparativa porque se ha decidido llevar a cabo esta decisión.

2.2.5 Tabla comparativa

Característica	NFS	LBFS	AFS	Coda	Solución que se va a estudiar
Uso transparente de red	✓	✓	✓	✓	✓
Consumo eficiente de ancho de banda	✗	✓	✓	✓	✓
Replicación	✗	✗	✓ (parcial)	✓	✓
Integración de módulos personalizables (p.e: logger)	✗	✗	✗	✗	✓

Tabla 1. Tabla comparativa de soluciones alternativas

2.3 Tecnologías

2.3.1 MQTT

MQTT es un protocolo de comunicación de publicación y suscripción. Cuenta principalmente con estos dos elementos:

- **Cliente MQTT:** se puede considerar como cliente a todo participante dentro de la aplicación MQTT que interviene en el proceso de intercambio de mensajes, y que es capaz tanto de enviar mensajes (publicar) como de recibirlos (suscribirse). Para ello, ha de estar conectado al Broker y tanto la publicación como la suscripción deben hacerse a unos temas específicos que se conocen como “tópicos”.
- **Broker MQTT:** El broker MQTT es el intermediario que se encarga de redirigir esos mensajes a través de los distintos clientes.

Para que el intercambio de mensajes se pueda realizar, los clientes MQTT hacen uso de diferentes callbacks. Entre los principales se encuentran:

Callback	Descripción
on_connect	Este callback se ejecuta cuando el cliente MQTT se conecta al Broker. Una vez llamado, se ejecuta la lógica que se implemente dentro de él.
on_publish	Este callback se ejecuta cuando el cliente MQTT publica un mensaje en uno de los tópicos.
on_message	Este callback se ejecuta cuando el cliente MQTT recibe algún mensaje en alguno de los tópicos a los que está suscrito.
on_subscribe	Este callback se ejecuta cuando el cliente MQTT se suscribe a algún tópico.
on_log	Este callback se ejecuta cuando se registra algún evento relacionado con el comportamiento de MQTT, principalmente sobre los eventos anteriores. Registra las operaciones que se hacen a la hora de publicar, etc.

Tabla 2. Definición de los callbacks de MQTT

Durante la publicación de un mensaje, es necesario enviar un argumento conocido como *Quality of Service* (QoS), y proporciona un mecanismo para evitar la pérdida de mensajes en los casos que se considere necesario.

Existen tres niveles:

Nivel	Descripción
Qos = 0	No se garantiza la recepción del mensaje por parte del suscriptor.
Qos = 1	El suscriptor le garantiza al publicador que el mensaje ha sido recibido como mínimo una vez, pero esto implica que el mensaje puede llegar repetido.
Qos = 2	Se garantiza que el mensaje es recibido por el suscriptor una única vez. No le pueden llegar mensajes repetidos.

Tabla 3. Explicación QoS de MQTT

Para la solución se va a utilizar el protocolo MQTT ya que proporciona los siguientes beneficios:

- **Bajo ancho de banda**

MQTT es un protocolo de mensajería ligero. Como el propósito de este trabajo es diseñar un sistema de ficheros de red donde el rendimiento sea óptimo aún en aquellas redes que no cuentan con un ancho de banda lo suficientemente amplio, es ideal incluirlo dentro de la arquitectura del proyecto.

- **Flexibilidad de la arquitectura publicación-suscripción**

Si aplicamos el protocolo a la solución que se quiere proyectar, MQTT puede ofrecer funcionalidades muy interesantes, tales como:

- Implementar un cliente MQTT que se encargue de la monitorización y análisis estadístico.
- Todos los clientes acceden a la versión más actualizada del contenido almacenado en el sistema de ficheros, en caso de que el sistema sea compartido entre varios clientes. Además es una forma sencilla de implementar este tipo de arquitectura.
- Cada cliente MQTT puede decidir a qué operaciones se suscribe, de tal modo que se puede introducir granularidad respecto a qué operaciones puede realizar cada uno dentro del sistema de ficheros.
- Se puede proporcionar redundancia mediante el uso de tantos clientes MQTT como se considere necesario. Esto hace que el sistema sea más fuerte al tener varios clientes MQTT con una copia del contenido, y será resistente frente a la pérdida de datos.

Básicamente, todas las ventajas se centran en el hecho de que tener varios clientes MQTT puede ayudar a introducir una capa de personalización muy modular y escalable dentro de la aplicación, donde cada cliente se encarga de una tarea distinta.

Existen varias posibles implementaciones del protocolo MQTT. En este caso se va a elegir Mosquitto, ya que es la más popular, y cuenta con una comunidad y soporte muy grandes.

2.3.2 FUSE (Filesystem in Userspace)

FUSE es una interfaz que permite crear sistemas de ficheros sin necesidad de editar el código del núcleo, para ello el código que implementa la lógica del sistema de archivos se ejecuta en el espacio de usuario, mientras que el módulo FUSE del kernel actúa como puente entre dicho proceso que se ejecuta en el espacio de usuario y el sistema operativo [11].

El siguiente esquema es de gran ayuda para explicar la arquitectura de FUSE:

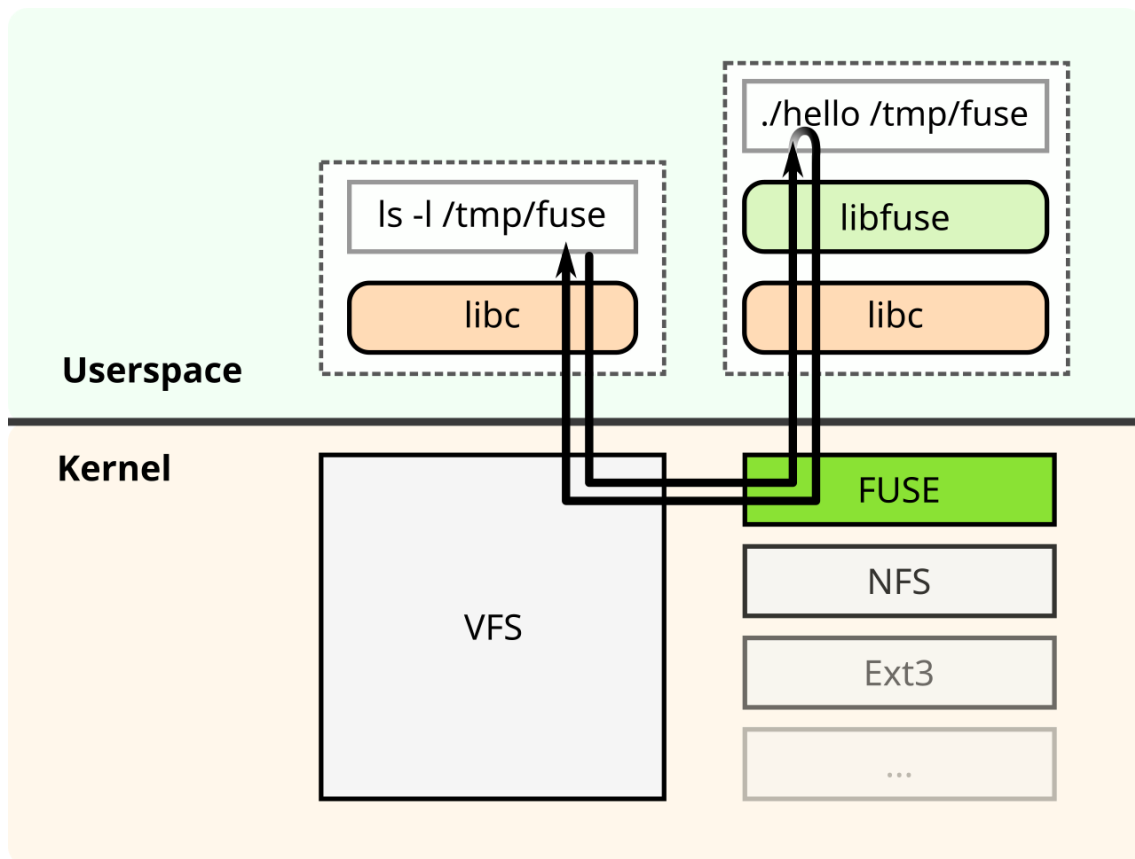


Figura 5. Arquitectura de FUSE [11]

Se van a explicar los pasos que ayudan a entender el flujo de su funcionamiento:

1. Primero el usuario ejecuta el programa “hello” con el directorio donde quiere montar el sistema de ficheros. El programa hello es el código con la implementación de FUSE, cuya interfaz es proporcionada mediante la librería libfuse.
2. El proceso llama al módulo FUSE en el kernel, y este a su vez se comunica con VFS para montar el sistema de ficheros.
3. Después el usuario ejecuta en el espacio de usuario cualquier operación relacionada con la gestión de ficheros. En el ejemplo se trata de un “ls -l /tmp/fuse/”
4. Esta llamada es interceptada por VFS en el espacio del kernel, que después es redirigida hacia el módulo FUSE del núcleo, y por último al programa “hello”
5. Aquí se ejecuta el código del sistema de ficheros personalizado, y se redirige la respuesta de nuevo al módulo FUSE de kernel.
6. La respuesta del módulo del kernel se envía a VFS y se ven los cambios reflejados en el espacio de usuario.

La librería que se va a usar para este proyecto es fusepy. Está librería se integra con el lenguaje de programación Python y proporciona una interfaz que permite utilizar libfuse mediante el uso de ctypes [12], librería que permite ejecutar y utilizar

estructuras propias del lenguaje de programación C. Por lo que fusepy actúa como un puente entre la utilización de la biblioteca libfuse con el uso del lenguaje Python.

2.3.3 ELK: Elasticsearch, Logstash y Kibana

La pila ELK está formada por las siguientes herramientas:

- Elasticsearch
- Logstash
- Kibana

Estas herramientas van a facilitar el proceso de almacenar, procesar, y visualizar los registros o logs. Mediante la transformación de los datos y el uso de filtros, se accede a la información relevante de una forma ordenada y estructurada. Gracias a la estructuración de esa información es posible la creación de distintos gráficos de interés. A continuación, se describe el propósito de cada una:

- **Logstash**

Esta herramienta se encarga de transformar los datos para su posterior almacenamiento y procesamiento. Gracias a esta herramienta se pueden insertar registros dentro de Elasticsearch con cualquier formato. Para ello es necesario implementar la lógica dentro del archivo de configuración.

Logstash se podría considerar una herramienta ETL (Extract, Transform, Load), ya que:

- Puede extraer la información de varias fuentes
- Permite transformar y limpiar los datos a través de diversos filtros
- Se pueden cargar los datos en múltiples destinos

En principio, se proporcionarán ficheros de logs como entrada, para poder extraer los campos de los mensajes presentes en los registros, y poder proporcionar la salida a Elasticsearch.

- **Elastic**

Como se explica en la página oficial de Elastic [13], Elasticsearch es un motor de búsqueda y análisis de datos distribuido. Es donde se produce la indexación, y proporciona análisis y búsqueda casi en tiempo real para todo tipo de datos. Elasticsearch puede almacenar cualquier tipo de datos, e indexarlos ofreciendo búsquedas rápidas.

- **Kibana**

Kibana es una herramienta de creación y visualización de datos. Se puede estructurar la forma en la que se muestran los mensajes de logs, y permite filtrar los mensajes en base a los campos presentes en cada registro.

Otra opción muy interesante es la visualización de los datos a través de los dashboards, donde se pueden crear numerosos gráficos y organizarlos para mostrar información que sea de interés.

El flujo de datos de la pila ELK es el siguiente:



Figura 6. Flujo de datos de ELK [14]

Se va a utilizar la pila ELK para proporcionar un entorno donde poder visualizar y ordenar la información de los ficheros log, los cuales se van a crear a través del cliente MQTT encargado de la monitorización.

2.3.4 Entorno de desarrollo

2.3.4.1 Linux Ubuntu 22.04

Sistema operativo basado en Linux y desarrollado por la empresa Canonical. Se ha elegido este sistema operativo debido a su mejor compatibilidad con las herramientas utilizadas en este proyecto, especialmente FUSE, ya que se integra de forma nativa. El resto de alternativas cuentan con más problemas de compatibilidad y no existe tanto soporte.

2.3.4.2 Python

Lenguaje de programación de alto nivel, interpretado y con tipado dinámico. Esto lo convierte en una opción muy flexible e intuitiva para poder desarrollar la solución estudiada. El IDE elegido para el desarrollo es Pycharm.

A través de la siguiente tabla se observa el estudio de soluciones alternativas que se ha realizado:

Métrica	Python	C++	Java
Soporte para MQTT	Sí (Eclipse Paho MQTT Python Client)	Sí (Eclipse Paho MQTT C++ Client)	Sí (Eclipse Paho Java Client)
Soporte para FUSE	Alto (fusepy)	Muy alto (libfuse)	Medio (jnr-fuse)
Facilidad de Uso	Muy alto	Medio/Bajo	Medio/Alto
Rendimiento	Medio/Bajo	Muy alto	Alto
Familiaridad	Muy alto	Medio	Alto
Soporte	Muy alto	Muy alto	Muy alto
Comunidad	Muy grande	Muy grande	Muy grande
Cantidad de bibliotecas	Muy alto	Muy alto	Muy alto

Tabla 4. Estudio de alternativas de lenguajes de programación

2.3.5 Bibliotecas utilizadas

Se describen en la siguiente tabla:

Nombre	Descripción
fusepy	Biblioteca que permite programar nuestro propio gestor de ficheros utilizando la interfaz FUSE.
paho-mqtt	Implementa la lógica de la comunicación del protocolo MQTT en Python.
json	Biblioteca que se utiliza para manejar archivos JSON (JavaScript Object Notation). Útil para serializar y deserializar los objetos Python que es necesario mandar a través de MQTT (en su mayoría diccionarios).
base64	Esta biblioteca se utiliza para la codificación de datos binarios a caracteres ASCII. Es necesario dado que existen situaciones donde el valor dentro de un par clave-valor es necesario que esté formado por una cadena de bytes, y los JSON no admiten este tipo de dato. También se valoró la opción de utilizar BSON (Binary JSON).

time	Sirve para implementar la espera activa que se utilizará en la función “sync” del código, y que sirve para sincronizar el lado cliente con la respuesta MQTT del servidor.
errno	Módulo que permite la utilización de constantes para los códigos de error.
os	Biblioteca que permite realizar las llamadas a bajo nivel relacionadas con la interfaz POSIX del sistema operativo, y que nos permite realizar las llamadas al kernel como read, write, getattr, etc.
Función “partial” del módulo “functools”	Se ha utilizado esta función para asignar determinados argumentos por defecto, debido a la necesidad de que se encuentren presentes en todas las llamadas. Por ejemplo, en la función “no_response_handler” que se verá más adelante, se utiliza esta función para pasar por defecto el objeto Cliente de la clase MQTT.
logging	Se utiliza para la monitorización y el pintado de trazas en ficheros log.
sys	Se utiliza para capturar los argumentos que se pasan por la línea de comandos. En este caso se captura el directorio donde se monta el sistema.
signal	Se utiliza para capturar la señal de interrupción del programa principal y llamar a la función fuse_exit() de fusepy. Esta función global se encarga de desmontar el sistema y cerrar la aplicación FUSE.

Tabla 5. Bibliotecas utilizadas en el proyecto

2.4 Marco regulador

En este capítulo se analiza la posible regulación que puede sufrir la aplicación bajo un contexto legislativo, la aplicación de normas y estándares que deben seguir las tecnologías que se van a usar para el desarrollo, y el uso de las licencias pertenecientes a las bibliotecas.

2.4.1 Marco legislativo

Debido a la naturaleza del proyecto, la única legislación que puede afectar al mismo es toda aquella relacionada con la protección de datos, al tener que tratar datos privados dentro de un almacenamiento alojado en la red. Este caso se daría siempre y cuando se decidiera comercializar la solución y el servidor no estuviese alojado por el propio cliente. Dicho esto, y poniéndonos en este supuesto caso, el proyecto debería cumplir con las siguientes legislaciones:

- **REGLAMENTO (UE) 2016/679 DEL PARLAMENTO EUROPEO Y DEL CONSEJO de 27 de abril de 2016 [15].**

Establece la normativa para la Unión Europea del tratado y la privacidad de los datos personales. Esta privacidad de los datos estaría relacionada con un proceso de registro de usuarios, donde se trata información personal, así como el tratamiento y almacenamiento de los propios ficheros que se alojasen dentro del servidor.

- **Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales.**

Esta ley es una adaptación que complementa al “*REGLAMENTO (UE) 2016/679*” para la legislación española. Tal y como se recoge en el artículo 82: “Los usuarios tienen derecho a la seguridad de las comunicaciones que transmitan y reciban a través de Internet” [16].

- **El apartado 18.4 de la constitución española.**

Tal y como establece dicho artículo: “La ley limitará el uso de la informática para garantizar el honor y la intimidad personal y familiar de los ciudadanos y el pleno ejercicio de sus derechos.” [17]

De nuevo, este artículo se puede interpretar como una regulación frente a la privacidad de los datos, en este caso relacionada con el uso de la informática, que engloba el uso de sistemas de ficheros y comunicaciones a través de Internet.

- **ISO/IEC 27001**

También se puede considerar seguir la norma “*ISO/IEC 27001*”, la cual establece unas prácticas y requisitos recomendados para evitar brechas de seguridad, y poder mantener la privacidad de los usuarios mediante la implementación de un sistema de gestión de seguridad de la información [18].

2.4.2 Estándares técnicos

Se utiliza la versión 3.1.1 del protocolo MQTT. Las reglas de esta versión 3.1.1, están recogidas en el estándar “*MQTT Version 3.1.1*” de la organización “OASIS” [19] (Organization for the Advancement of Structured Information Standards), que después sería ratificado por la norma “*ISO/IEC 20922:2016*” [20].

Otra librería ampliamente utilizada en este proyecto es JSON, cuyo estándar se recoge en “*ISO/IEC 21778:2017*”. [21]

2.4.3 Uso de licencias

Según el repositorio oficial de libfuse [22], se utiliza una licencia “*GNU Lesser General Public License (LGPL)*” para todos los ficheros contenidos en los directorios

“include/”, y “lib/”. El resto de ficheros solo se puede utilizar bajo una licencia “*GNU General Public License (GPL)*”. Se distribuye LGPL en su versión 2.1 y GPL en su versión 2.

GPL permite compartir y modificar el software siempre y cuando se distribuya bajo la misma licencia GPL. En el caso de LGPL, es más permisiva y no requiere obligatoriamente que se use una licencia GPL/LGPL [23].

La biblioteca fusepy se distribuye bajo una licencia “*ISC License*”. Esta licencia es muy permisiva ya que permite usar, modificar y distribuir software, siempre y cuando se incluya esta licencia y el aviso de copyright en el proyecto donde se está utilizando dicho software [24].

La biblioteca paho-mqtt hace uso de la licencia “*Eclipse Public License 2.0*” y de la licencia “*Eclipse Distribution License 1.0*”, como se puede observar en el archivo de texto “LICENSE.txt” dentro del repositorio oficial en GitHub. [25]

El resto de librerías al formar parte de la biblioteca estándar de Python cuentan con una licencia PSF (*Python Software Foundation License*). [26]

2.5 Impacto socioeconómico

A continuación se va a realizar un análisis del impacto de este proyecto en diversas áreas, con el fin de analizar su influencia en diversos ámbitos.

2.5.1 Impacto social

Este proyecto se realiza con la intención de poder ofrecer un sistema de ficheros distribuido fácil de implementar. Y de hecho así es, ya que únicamente se requiere de al menos dos máquinas Linux con Python instalado, y que cuente con un alojamiento en la red para poder alojar los procesos. Esto montará el sistema de ficheros con total transparencia, aparentando que el sistema se ejecuta localmente, por lo que el usuario no se tiene que preocupar de ningún aspecto técnico.

Por otro lado, el ser una solución de tan bajo coste favorecerá la implementación del mismo, ya que es una solución de código abierto que no requiere ninguna suscripción ni pago más que el del propio servidor.

El protocolo MQTT, característico por su eficiencia en el consumo de ancho de banda, permite que el uso del sistema de ficheros sea conveniente en aquellas infraestructuras tecnológicamente limitadas, como puede ser el caso de zonas rurales, donde la cobertura de las redes inalámbricas puede ser bastante reducida.

2.5.2 Impacto económico

La versatilidad y modularidad que ofrece el diseño mediante clientes MQTT convierte este sistema de ficheros en una solución atractiva para las empresas, puesto que se pueden adaptar necesidades y servicios específicos.

El uso de estos clientes MQTT se puede considerar una gran solución económica al problema de la escalabilidad horizontal que pueda sufrir una organización cuando sus necesidades de almacenamiento aumentan, ya que teóricamente se puede distribuir la carga entre varios nodos mediante la implementación de varios clientes, y también ofrece características atractivas como tolerancia a fallos mediante la replicación de nodos.

Como MQTT es un protocolo de mensajería ligero, esto puede favorecer la eficiencia a la hora de comunicar los datos, y teóricamente puede repercutir en numerosas ventajas:

- Por un lado los mensajes MQTT cuentan con un menor *overhead* (datos de control), por lo que se espera un menor procesamiento, lo que en última instancia hace que se recorten gastos en computación.
- Dicha eficiencia permite recortar ligeramente los gastos en hardware, ya que se aprovecha mejor el equipo para procesar la misma respuesta.
- Todo esto tiene como consecuencia un menor coste energético.
- Por último, el hecho de que MQTT utilice menos ancho de banda reduce la carga de red, optimizando el uso de esta y reduciendo los costes de conexión.

2.5.3 Impacto medioambiental

Debido a las razones anteriormente expuestas en el apartado económico, hay una repercusión indirecta del uso de MQTT, donde su característica eficiencia en el consumo de ancho de banda, produce que se optimicen los recursos computacionales y de hardware, y por tanto se reduzca el coste energético.

3. ANÁLISIS, DISEÑO, IMPLEMENTACIÓN E IMPLANTACIÓN

3.1 Metodología

Para la realización de este trabajo, el desarrollo va a seguir un ciclo de vida en cascada.

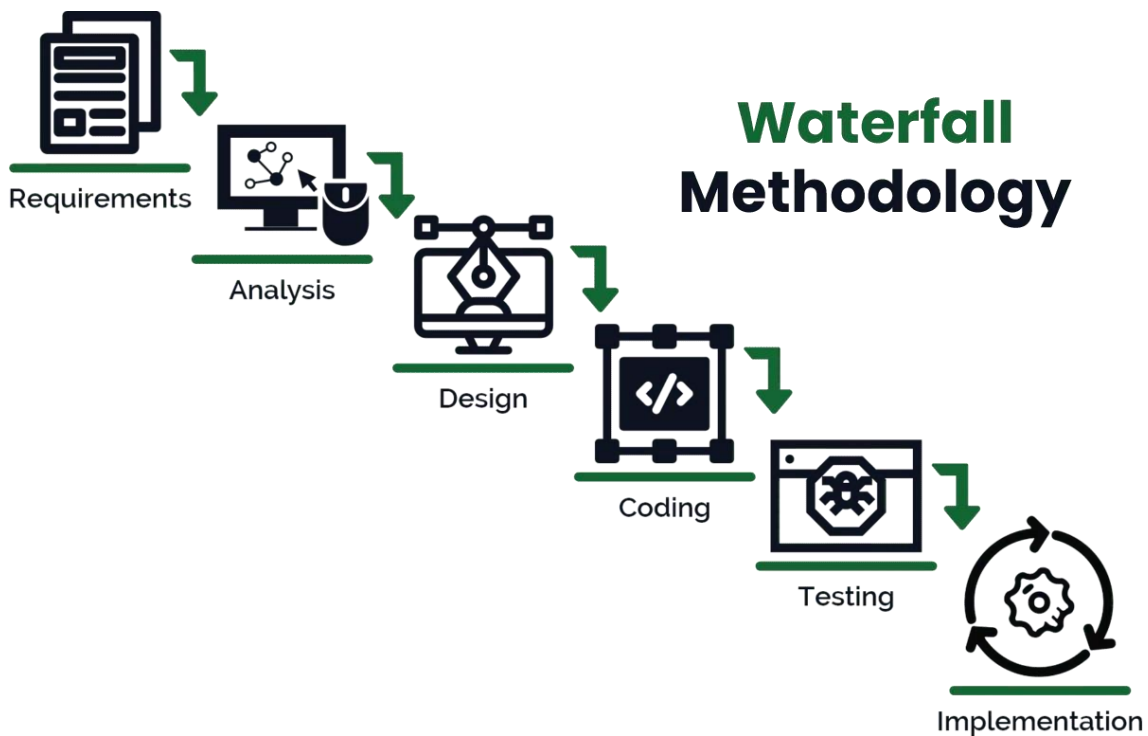


Figura 7. Ciclo de vida en cascada [27]

Como se puede ver en este esquema, se sigue una estructura de fases fija y con una secuencia lineal. Una vez completada una fase no se vuelve a fases anteriores. Por este motivo, requiere una planificación extensa, y solo es apta para aquellos proyectos donde están muy bien definidos los requisitos y que no requieran de ningún cambio que se salga de lo planificado a lo largo de la vida del proyecto.

Esta es la razón por la que se ha elegido esta metodología, puesto que al ser yo el *stakeholder* junto con mi tutor Alejandro, y al ser el desarrollador de la solución, se tiene una idea muy clara de qué se va a necesitar, y qué va a requerir la solución para poder cumplir los objetivos iniciales. Además, al contrario que en la mayoría de las metodologías que tienen en cuenta el trabajo en equipo y el reparto de tareas en paralelo, este trabajo ha sido realizado únicamente por una persona, por lo que un desarrollo secuencial puede ser más lógico e intuitivo.

Como se observa en el **apartado 6.1**, la planificación para este ciclo de vida es esencial. A diferencia de las metodologías ágiles donde el desarrollo suele ser iterativo e incremental, y razón por la cual se permite mucha más flexibilidad, aquí el proyecto se encuentra restringido por unos plazos y etapas que se deben realizar estrictamente.

A continuación, voy a definir las etapas que conforman el proyecto:

1. Análisis

Durante esta etapa inicial, se analizan los requerimientos de la solución que se va a desarrollar. Una vez realizado este proceso, se documentan y definen los requisitos de la forma más rigurosa y formal posible para que no haya espacio a interpretaciones erróneas. Se van a definir tanto requisitos funcionales como no funcionales. También se diseñan los casos de uso. Esto forma parte del **apartado 3.2.**

2. Diseño

En la etapa de **Diseño**, se incluyen los siguientes diagramas que ayudan a tener una visión del concepto que se quiere desarrollar en la etapa posterior:

- Diagramas de secuencia
- Diagrama de clases
- Diagrama de arquitectura del sistema

3. Desarrollo

Durante esta fase se codifica la solución en base a los requisitos y diseño de las etapas anteriores. Este proceso ha sido documentado en el **apartado 3.4.**

Se va a explicar para qué sirven todos los scripts y módulos presentes en el proyecto, el propósito de todas las funciones involucradas, la estructura del proyecto, y por último se explica qué partes del código han supuesto una dificultad en su implementación.

4. Pruebas (Testing)

Sobre el código desarrollado en la etapa anterior, se realizan las debidas pruebas que garanticen que se cumplen los requisitos de la primera etapa. Este proceso se recoge en el capítulo de **Evaluación.**

5. Despliegue

Se va a explicar cómo se instalan las dependencias necesarias para que funcione la solución, y cómo se despliega. Esta etapa se recoge en el epígrafe de **Implantación.**

3.2 Análisis

En este capítulo se va a desarrollar el proceso de “Análisis”. Este proceso es primordial, debido a que en él se identifican y definen las necesidades y objetivos que hay que cumplir para llegar a una solución satisfactoria. En esta etapa participa tanto el creador de la solución como los *stakeholders*, que en este caso sería únicamente el tutor a cargo de este Trabajo de fin de Grado.

Por otro lado, se va a detallar la solución que se ha ideado y se quiere desarrollar, el proceso a seguir para la recopilación de requisitos, su especificación, el desarrollo de los casos de uso, y se van a comparar en una matriz de trazabilidad para una mejor comprensión y seguimiento. Por último se va a profundizar en la importancia de este proceso.

Una buena especificación de requisitos va a facilitar mucho el trabajo futuro, ya que te permite entender mejor el proyecto, y a tener una visión global del mismo y específica de cada componente que lo compone. También evita contratiempos fruto de posibles malentendidos, ayudando con la planificación, y estableciendo mejor las necesidades que se requieren.

Además, es importante recalcar que este proceso es de vital importancia realizarlo exhaustivamente en el ciclo de vida en cascada que se ha elegido, ya que en este ciclo de vida se desarrollan las etapas secuencialmente, y no entra dentro de la planificación volver a etapas anteriores, por lo que un fallo en esta etapa de análisis puede provocar retrasos y cambios en la planificación.

3.2.1 Descripción detallada de la solución

Después de haber presentado la información necesaria para entender el proyecto dentro del capítulo del **Estado de la cuestión**, se puede explicar la solución proyectada.

El esqueleto del sistema de ficheros va a estar compuesto por dos ficheros Python (sin contar las funcionalidades de monitorización y replicación), uno que se va a ejecutar en la máquina local del cliente, y el otro fichero va a estar alojado en el lado servidor.

El código que se ejecuta en el lado cliente va a montar el sistema de ficheros en el espacio de usuario. Este programa se ejecuta por terminal junto con un argumento, el cual va a indicar el punto de montaje. Esto monta el directorio en el sistema operativo, donde se va a poder trabajar con los distintos ficheros, como si se encontrasen localmente en la máquina, ya sea interactuando con dichos ficheros a través de la interfaz gráfica, o a través de la terminal.

Por otro lado tenemos el ejecutable del servidor, que va a interactuar con los ficheros que se encuentran en el almacenamiento de la máquina. Para poder interactuar con el almacenamiento del sistema operativo se utiliza la librería de Python “os”.

Ambos componentes tienen que comunicarse para poder realizar el intercambio de solicitud y respuesta, que en este caso se hace mediante MQTT. Para cada operación que se ha desarrollado para el sistema de ficheros, se han creado dos tópicos:

- Un “**REQUEST_OPERATION_TOPIC**”

El cliente publica la información que requiere del servidor en un diccionario (el cual se ha de serializar antes de hacer el *publish*, ya que solo se pueden mandar mensajes en binario). El servidor, el cual está suscrito a este tópico, recibe la información dentro de la función “on_message” a través del objeto *msg*. Se deserializa para más tarde pasar los argumentos que se encuentran dentro del diccionario a la operación de la librería “os” correspondiente.

- Un “**OPERATION_TOPIC**”

El servicio que se encuentra alojado en el servidor manda el resultado de las operaciones del sistema de ficheros a través de este tópico, al cual estará suscrito el cliente MQTT. En caso de error se manda el código para que se haga el manejo de excepciones en el lado cliente.

Por otro lado el cliente MQTT encargado de guardar los registros en un fichero log escuchará en el tópico “**LOGGING_FS_TOPIC**”. Este cliente va a almacenar los distintos eventos dentro del sistema en dos ficheros diferentes:

- **logging_FUSE.log**: registra los eventos relacionados con el procesamiento de las operaciones del sistema de ficheros.
- **logging_MQTT.log**: registra los eventos relacionados con la comunicación MQTT.

Después estos ficheros se proporcionan como entrada a Logstash, que almacena la información en Elasticsearch, para poder visualizar después la información con Kibana.

3.2.2 Requisitos

Antes de proceder con la definición de requisitos, es necesario aclarar que la obtención de estos ha surgido por un lado de una sesión de *Brainstorming*, y por otro lado de las reuniones con el tutor, que en este caso se podría considerar un *stakeholder*.

Se va a utilizar una plantilla de tipo tabla para la definición de requisitos y es la siguiente:

Identificador del requisito	
Nombre	Título que describe el requisito
Prioridad	Alta / Media / Baja
Verificabilidad	Alta / Media / Baja
Fecha	Fecha de creación del requisito
Descripción	Descripción breve del requisito

Tabla 6. Plantilla de requisitos

Como se puede observar en la plantilla, los campos que van a describir formalmente cada uno de los requisitos son los siguientes:

- **Identificador del requisito:** Este campo es un “ID” que localiza unívocamente cada requisito. Se va a seguir la siguiente convención:
 - “RF-XX” / “RNF-XX”
 - **RF:** “Requisitos Funcionales”
 - **RNF:** “Requisitos No Funcionales”
 - **XX:** Número que identifica el requisito, en orden de aparición
- **Nombre:** El nombre que se le va a dar a cada requisito debe ser lo más descriptivo posible para ayudar con la identificación durante la fase de desarrollo.
- **Prioridad:** Ayuda a establecer la urgencia con la que se deben implementar los requisitos. Se va a seguir la siguiente convención:
 - Alta
 - Media
 - Baja
- **Verificabilidad:** Un requisito es verificable cuando se puede probar de forma objetiva que cumple su objetivo. Ayuda a eliminar la ambigüedad al definir los requisitos, en el caso de ser posible. Los posibles valores de este campo son:
 - Alta
 - Media
 - Baja
- **Fecha:** La fecha de creación del requisito. Se va a seguir el formato: “dd/mm/yy”
- **Descripción:** Comentario que va a describir el objetivo del requisito de manera precisa y concisa, para no dar pie a ambigüedades.

3.2.2.1 Requisitos funcionales

Se va a proceder primero con la definición de los requisitos funcionales.

RF-01	
Nombre	Ubicación del punto de montaje
Prioridad	Alta
Verificabilidad	Alta
Fecha	22/02/24
Descripción	El punto de montaje del sistema de ficheros será elegido por el usuario, y podrá ser introducido como argumento a través de la terminal. Concretamente es el primer y único argumento.

Tabla 7. RF-01

RF-02	
Nombre	Tipo de sistema de ficheros
Prioridad	Alta
Verificabilidad	Alta
Fecha	22/02/24
Descripción	Se va a implementar un sistema de ficheros de red. El usuario no tendrá ninguno de sus ficheros en el almacenamiento de su máquina local, pero a través de una interfaz podrá acceder a ellos y manipularlos. Estos se encontrarán alojados en un servidor.

Tabla 8. RF-02

RF-03	
Nombre	Compatibilidad del manejo de archivos
Prioridad	Alta
Verificabilidad	Alta
Fecha	22/02/24
Descripción	El usuario podrá realizar la manipulación de ficheros (abrir, escribir, leer, etc.) a través de la interfaz gráfica del SO, y por terminal.

Tabla 9. RF-03

RF-04	
Nombre	Apertura de ficheros
Prioridad	Alta
Verificabilidad	Alta
Fecha	23/02/24
Descripción	El usuario podrá abrir cualquier tipo de fichero, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 10. RF-04

RF-05	
Nombre	Lectura de ficheros
Prioridad	Alta
Verificabilidad	Alta
Fecha	23/02/24
Descripción	El usuario podrá realizar lecturas sobre cualquier tipo de fichero, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 11. RF-05

RF-06	
Nombre	Escritura de ficheros
Prioridad	Alta
Verificabilidad	Alta
Fecha	23/02/24
Descripción	El usuario podrá realizar escrituras sobre cualquier tipo de fichero, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 12. RF-06

RF-07	
Nombre	Cierre de ficheros
Prioridad	Alta
Verificabilidad	Alta
Fecha	24/02/24
Descripción	El usuario podrá cerrar cualquier tipo de fichero, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 13. RF-07

RF-08	
Nombre	Cambio de nombre de ficheros
Prioridad	Baja
Verificabilidad	Alta
Fecha	24/02/24
Descripción	El usuario podrá renombrar cualquier tipo de fichero, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 14. RF-08

RF-09	
Nombre	Creación de ficheros
Prioridad	Media
Verificabilidad	Alta
Fecha	24/02/24
Descripción	El usuario podrá crear cualquier tipo de fichero, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 15. RF-09

RF-10	
Nombre	Eliminación de ficheros
Prioridad	Alta
Verificabilidad	Alta
Fecha	24/02/24
Descripción	El usuario podrá borrar cualquier tipo de fichero, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 16. RF-10

RF-11	
Nombre	Modificar los permisos de los ficheros
Prioridad	Media
Verificabilidad	Alta
Fecha	24/02/24
Descripción	<p>El usuario podrá modificar los permisos de cualquier tipo de fichero, en cualquier subdirectorío del punto de montaje, siempre que el usuario sea el propietario del fichero. El usuario podrá cambiar los permisos de lectura, escritura y ejecución para:</p> <ul style="list-style-type: none"> • El propietario • El grupo propietario • “Otros”

Tabla 17. RF-11

RF-12	
Nombre	Comprobación de permisos
Prioridad	Alta
Verificabilidad	Alta
Fecha	26/02/24
Descripción	El sistema de ficheros comprobará que el usuario tiene los permisos necesarios antes de poder manipular cualquier fichero que se encuentre dentro del punto de montaje.

Tabla 18. RF-12

RF-13	
Nombre	Apertura de directorios
Prioridad	Alta
Verificabilidad	Alta
Fecha	26/02/24
Descripción	El usuario podrá abrir cualquier directorio, en cualquier subdirectorio del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 19. RF-13

RF-14	
Nombre	Creación de directorios
Prioridad	Media
Verificabilidad	Alta
Fecha	26/02/24
Descripción	El usuario podrá crear cualquier directorio, en cualquier subdirectorio del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 20. RF-14

RF-15	
Nombre	Eliminación de directorios
Prioridad	Baja
Verificabilidad	Alta
Fecha	26/02/24
Descripción	El usuario podrá eliminar cualquier directorio, en cualquier subdirectorio del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 21. RF-15

RF-16	
Nombre	Cambiar el nombre de un directorio
Prioridad	Baja
Verificabilidad	Alta
Fecha	27/02/24
Descripción	El usuario podrá renombrar cualquier directorio, en cualquier subdirectorio del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 22. RF-16

RF-17	
Nombre	Listar los ficheros de un directorio
Prioridad	Alta
Verificabilidad	Alta
Fecha	27/02/24
Descripción	El usuario podrá listar los contenidos de cualquier directorio, en cualquier subdirectorio del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 23. RF-17

RF-18	
Nombre	Obtener los atributos de un fichero
Prioridad	Alta
Verificabilidad	Alta
Fecha	27/02/24
Descripción	<p>El usuario podrá obtener los atributos de cualquier fichero, en cualquier subdirectorio del punto de montaje, siempre que el usuario tenga los permisos necesarios. Se entregan los siguientes atributos:</p> <ul style="list-style-type: none"> • Los permisos de lectura, escritura y ejecución para el usuario propietario, grupo propietario, y “otros” • Tipo de fichero (fichero regular, directorio, enlace simbólico, etc.) • Usuario propietario • Grupo propietario • Número de enlaces • Tamaño del fichero

Tabla 24. RF-18

RF-19	
Nombre	Crear enlaces “duros”
Prioridad	Baja
Verificabilidad	Alta
Fecha	28/02/24
Descripción	El usuario podrá crear enlaces “duros”, en cualquier subdirectorio del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 25. RF-19

RF-20	
Nombre	Crear enlaces “blandos”
Prioridad	Baja
Verificabilidad	Alta
Fecha	28/02/24
Descripción	El usuario podrá crear enlaces “blandos”, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 26. RF-20

RF-21	
Nombre	Abrir enlaces “duros” y “blandos”
Prioridad	Baja
Verificabilidad	Alta
Fecha	28/02/24
Descripción	El usuario podrá abrir tanto enlaces “blandos” como enlaces “duros”, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 27. RF-21

RF-22	
Nombre	Eliminar enlaces “duros” y “blandos”
Prioridad	Baja
Verificabilidad	Alta
Fecha	28/02/24
Descripción	El usuario podrá borrar tanto enlaces “blandos” como enlaces “duros”, en cualquier subdirectorío del punto de montaje, siempre y cuando el usuario tenga los permisos necesarios.

Tabla 28. RF-22

RF-23	
Nombre	Casos de error
Prioridad	Alta
Verificabilidad	Media
Fecha	29/02/24
Descripción	En caso de cualquier tipo de error, el sistema notificará dicho error al usuario con el mismo mensaje que recibiría en el sistema de ficheros nativo de su SO.

Tabla 29. RF-23

RF-24	
Nombre	Implementación de un logger
Prioridad	Alta
Verificabilidad	Alta
Fecha	29/02/24
Descripción	<p>Uno de los clientes MQTT actúa como <i>logger</i>. El propósito es guardar un registro exhaustivo de toda la información relacionada con el sistema de ficheros. Esta monitorización va a estar almacenada en ficheros log. Se van a registrar tres tipos de niveles y la siguiente información en cada uno de ellos:</p> <ul style="list-style-type: none"> • DEBUG <ul style="list-style-type: none"> ○ Tópico ○ Función ○ Argumentos ○ Resultado de la operación • INFO <ul style="list-style-type: none"> ○ Tópico • ERROR <ul style="list-style-type: none"> ○ Función ○ Argumentos ○ Resultado de la operación ○ Código de error ○ Mensaje de error

Tabla 30. RF-24

RF-25	
Nombre	Tolerancia a fallos
Prioridad	Alta
Verificabilidad	Alta
Fecha	29/02/24
Descripción	<p>Se va a implementar mínimo un cliente MQTT que actúe a modo de “nodo de replicación” para proporcionar un mecanismo de tolerancia a fallos para el sistema. Este cliente MQTT registrará todas las operaciones de escritura escuchando en el mismo tópico que el cliente principal que gestiona las peticiones.</p>

Tabla 31. RF-25

RF-26	
Nombre	Herramienta de monitorización y visualización de datos
Prioridad	Baja
Verificabilidad	Alta
Fecha	29/02/24
Descripción	<p>Se debe proporcionar una herramienta que permita la visualización de los datos contenidos en los ficheros de registro.</p>

Tabla 32. RF-26

RF-27	
Nombre	Desmontar el sistema de ficheros
Prioridad	Media
Verificabilidad	Alta
Fecha	29/02/24
Descripción	El sistema de ficheros se podrá desmontar desde la terminal con el comando “fusermount” y el parámetro “-u”, o bien mediante la interrupción de la aplicación.

Tabla 33. RF-27

3.2.2.2 Requisitos no funcionales

RNF-01	
Nombre	Transparencia de red
Prioridad	Alta
Verificabilidad	Alta
Fecha	29/02/24
Descripción	El sistema de ficheros debe implementarse con transparencia de red, es decir, se accederá al sistema de ficheros en la máquina cliente como si fuera un sistema de ficheros local.

Tabla 34. RNF-01

RNF-02	
Nombre	Python
Prioridad	Alta
Verificabilidad	Alta
Fecha	29/02/24
Descripción	Todo el código va a ser desarrollado en Python

Tabla 35. RNF-02

RNF-03	
Nombre	Implementación de red con MQTT
Prioridad	Alta
Verificabilidad	Alta
Fecha	29/02/24
Descripción	Para implementar la comunicación entre el cliente y el servidor se va a utilizar el protocolo de comunicación MQTT. Concretamente se va a hacer uso de la librería paho-mqtt.

Tabla 36. RNF-03

RNF-04	
Nombre	Implementación del sistema de ficheros con FUSE
Prioridad	Alta
Verificabilidad	Alta
Fecha	01/03/24
Descripción	El sistema de ficheros se va a implementar con FUSE. Concretamente, se va a hacer uso de la librería <i>fusepy</i> .

Tabla 37. RNF-04

RNF-05	
Nombre	Linux
Prioridad	Media
Verificabilidad	Baja
Fecha	01/03/24
Descripción	El sistema de ficheros que se va a implementar va a ser compatible con cualquier distribución Linux.

Tabla 38. RNF-05

RNF-06	
Nombre	Interfaz de bajo nivel para las llamadas al sistema operativo
Prioridad	Alta
Verificabilidad	Alta
Fecha	01/03/24
Descripción	Para la correcta implementación del sistema de ficheros, se va a utilizar una biblioteca que permita interactuar con el sistema operativo a bajo nivel. En ese caso se ha optado por usar la biblioteca os.

Tabla 39. RNF-06

RNF-07	
Nombre	Serialización con <i>JSON</i>
Prioridad	Alta
Verificabilidad	Alta
Fecha	01/03/24
Descripción	Se utilizará la biblioteca JSON para la serialización de los datos que se envíen a través de MQTT.

Tabla 40. RNF-07

RNF-08	
Nombre	Código limpio
Prioridad	Alta
Verificabilidad	Baja
Fecha	01/03/24

Descripción	El código va a estar lo más modularizado y legible posible. Esto permite un mejor mantenimiento del código.
--------------------	---

Tabla 41. RNF-08

RNF-09	
Nombre	Código comentado
Prioridad	Alta
Verificabilidad	Alta
Fecha	01/03/24
Descripción	Todo el código debe estar debidamente comentado para su correcto entendimiento por parte de terceras personas. Las explicaciones deben aclarar totalmente el funcionamiento de las funciones, siendo lo más concisas posible.

Tabla 42. RNF-09

RNF-10	
Nombre	Librería para implementar los loggings
Prioridad	Alta
Verificabilidad	Alta
Fecha	01/03/24
Descripción	El cliente MQTT que escribe los mensajes de monitorización en los ficheros log se va a implementar con la biblioteca de Python llamada <i>logging</i> .

Tabla 43. RNF-10

RNF-11	
Nombre	Stack ELK
Prioridad	Baja
Verificabilidad	Alta
Fecha	29/02/24
Descripción	La información de los ficheros log se va a almacenar y procesar a través del stack ELK.

Tabla 44. RNF-11

3.2.3 Casos de uso

A través del diagrama de casos de uso, se va a analizar la interacción entre el usuario y el sistema. Este proceso ayuda en la elicitación de requisitos funcionales, y a entender mejor el comportamiento general del proyecto desde la perspectiva del usuario.

Para ello, es necesario identificar los actores y sistemas involucrados, y la secuencia de acciones que realiza un usuario y que provoca una modificación dentro del sistema.

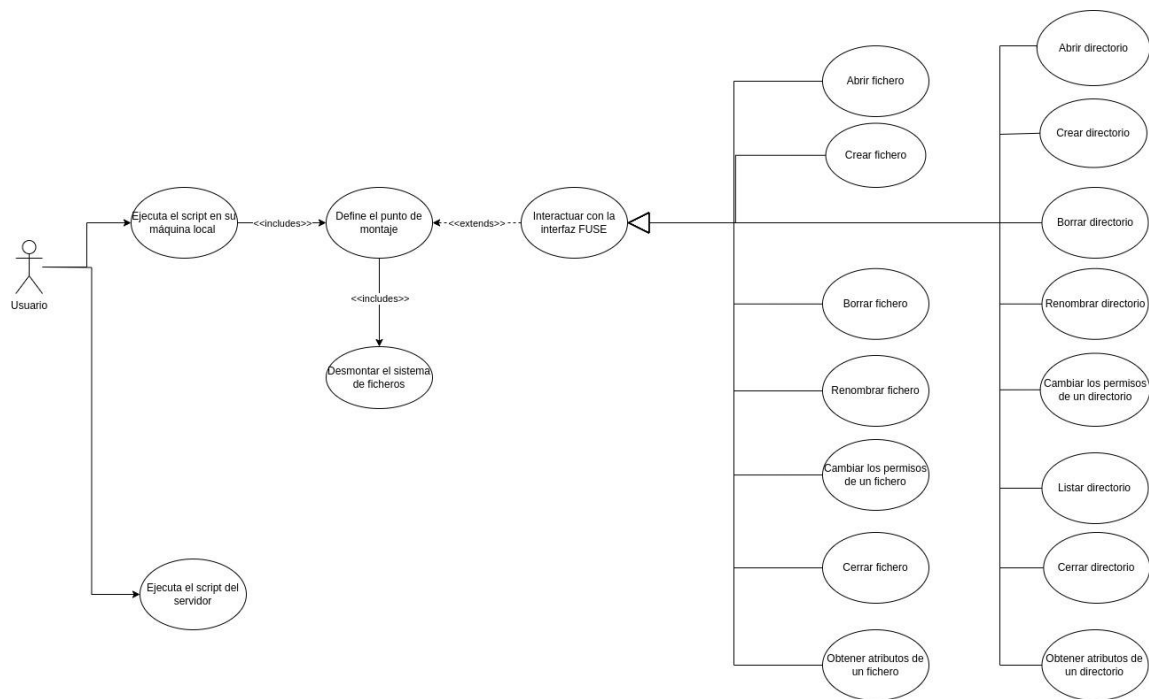


Figura 8. Diagrama de casos de uso

En este caso solo tenemos un único usuario. Se consideró la opción de incluir como actores el “Sistema Operativo del lado servidor”, ya que se encarga del manejo de ficheros, y el “Broker MQTT”. Sin embargo, las acciones del servidor son consecuencia de las acciones realizadas por el usuario en el lado cliente, por lo que se ha considerado que estos dos agentes forman parte del comportamiento general del sistema, más que como dos actores independientes y externos al mismo.

A continuación, se va a realizar una descripción formal de cada caso de uso dentro del diagrama, siguiendo la misma metodología que con los requisitos. La plantilla que se va a utilizar va a ser la siguiente:

CU-XX	
Nombre	
Precondiciones	
Escenario	
Postcondiciones	
Dependencias	

Tabla 45. Plantilla de casos de uso

Los campos que componen la tabla son los siguientes:

- **Identificador del caso de uso:** Este campo es un “ID” que localiza unívocamente cada caso de uso. Se va a seguir la siguiente convención:
 - “CU-XX”
 - “CU”: “Caso de uso”
 - “XX”: Número que identifica el caso de uso, en orden de aparición
- **Nombre:** El nombre que se le va a dar a cada caso de uso debe ser lo más descriptivo posible.
- **Precondiciones:** Lista de condiciones que deben cumplirse para que se pueda llevar a cabo la realización del caso de uso.
- **Escenario:** El flujo de acciones que se realizan bajo el uso habitual del sistema.
- **Postcondiciones:** Se muestran los efectos de la ejecución de las acciones que caracterizan el caso de uso.
- **Dependencias:** En este campo se muestran otros casos de uso que dependen del actual.

CU-01	
Nombre	Ejecutar el script del lado cliente
Precondiciones	<ul style="list-style-type: none"> • Tener un ordenador con Sistema Operativo Linux. • Es necesario tener instalado Python, y todas las dependencias necesarias. • El Broker MQTT tiene que ejecutarse previamente para poder iniciar la comunicación. • Es necesario ejecutar primero el script del servidor.
Escenario	<ol style="list-style-type: none"> 1. El usuario localiza la ubicación del script dentro de su sistema operativo. 2. El usuario ejecuta el script pasando la ubicación del ejecutable como primer argumento, y la ubicación donde se va a montar el sistema de ficheros como segundo argumento.
Postcondiciones	<ul style="list-style-type: none"> • Se monta el sistema de ficheros FUSE en el directorio que el usuario ha elegido.
Dependencias	Ninguna

Tabla 46. CU-01

CU-02	
Nombre	Definir el punto de montaje
Precondiciones	Tener el código localizado en la máquina donde se va a decidir ejecutarlo.
Escenario	<ol style="list-style-type: none"> 1. El usuario ejecuta el código. 2. Pasa como argumento el punto de montaje.
Postcondiciones	<ul style="list-style-type: none"> • El sistema de ficheros se ha montado y ya se puede interactuar con el mismo.
Dependencias	CU-01

Tabla 47. CU-02

CU-03	
Nombre	Ejecutar el script del lado servidor
Precondiciones	<ul style="list-style-type: none"> • Tener un servidor Linux donde poder alojar los datos de los ficheros, y poder ejecutar el script. • El Broker MQTT tiene que ejecutarse previamente para poder iniciar la comunicación.
Escenario	<ol style="list-style-type: none"> 1. El usuario ejecuta el script.
Postcondiciones	<ul style="list-style-type: none"> • El cliente MQTT instalado en el servidor estará escuchando en los tópicos correspondientes esperando a recibir las órdenes que ejecutar en el kernel del SO.
Dependencias	Ninguna

Tabla 48. CU-03

CU-04	
Nombre	Interactuar con la interfaz FUSE
Precondiciones	<ul style="list-style-type: none"> • Haber montado el sistema de ficheros. • Haber ejecutado el script del lado servidor. • Tener los permisos necesarios para interactuar con los ficheros y directorios. • Estar ubicado dentro del punto de montaje.
Escenario	El usuario puede utilizar el sistema de ficheros con todas aquellas operaciones implementadas en el código a través de la interfaz FUSE (escribir, leer, cerrar ficheros, etc.).
Postcondiciones	<ul style="list-style-type: none"> • El usuario puede interactuar con el sistema de ficheros distribuido.
Dependencias	CU-02, CU-03

Tabla 49. CU-04

CU-05	
Nombre	Desmontar el sistema de ficheros
Precondiciones	<ul style="list-style-type: none"> Tener montado el sistema de ficheros previamente.
Escenario	1. El cliente desmontará el sistema de ficheros por terminal haciendo uso del comando “fusermount” y la opción “-u”, o parando la ejecución del script.
Postcondiciones	<ul style="list-style-type: none"> El sistema de ficheros se ha desmontado, y ya no es posible hacer uso del mismo. El lado servidor se quedará encendido a no ser que se decida desconectarlo.
Dependencias	CU-02

Tabla 50. CU-05

Debido a la similitud entre todos los casos de uso relacionados con la gestión del sistema de ficheros, se ha decidido poner el caso de uso general: “*Interactuar con la interfaz FUSE*”. Añadir estos casos de uso no aporta ninguna información adicional además de especificar la operación que se ha de ejecutar.

3.2.4 Matriz de trazabilidad

A continuación, se va a estudiar la relación entre los requisitos funcionales y los casos de uso a través de la siguiente tabla:

	CU-01	CU-02	CU-03	CU-04	CU-05
RF-01	X	X			
RF-02			X		
RF-03				X	
RF-04				X	
RF-05				X	
RF-06				X	
RF-07				X	
RF-08				X	
RF-09				X	
RF-10				X	
RF-11				X	
RF-12				X	
RF-13				X	
RF-14				X	
RF-15				X	
RF-16				X	
RF-17				X	
RF-18				X	
RF-19				X	
RF-20				X	
RF-21				X	
RF-22				X	
RF-23				X	
RF-24					
RF-25					

RF-26					
RF-27					X

Tabla 51. Matriz de trazabilidad I

Como se puede observar, la mayoría de los requisitos están englobados dentro del caso de uso “*CU-04*”, ya que este último es fruto de una generalización que abarca otros casos de uso. Los requisitos “*RF-24*”, “*RF-25*”, y “*RF-26*” no tienen correspondencia directa con ningún caso de uso al ser sistemas con los que el usuario no interactúa, concretamente el *logger* y el cliente de replicación.

3.3 Diseño

Esta etapa parte del proceso de ingeniería de requisitos que se realizó en el capítulo anterior, y sirve como base para la siguiente etapa del ciclo de vida en cascada, la implementación.

Para ello se va a estudiar la solución a través de distintos diagramas UML que van a mejorar la comprensión del sistema, y que agilizarán el proceso de desarrollo de código. Concretamente se va a estudiar la arquitectura general del sistema y sus distintos componentes, la interacción entre ellos, la estructura del código a través de sus distintas funciones y atributos, y la interacción en el tiempo entre los distintos módulos. Para ello se va a diseñar un diagrama de arquitectura del sistema, un diagrama de clases, y diagramas de secuencia.

3.3.1 Arquitectura del sistema

A través del siguiente diagrama de alto nivel se van a presentar los distintos componentes que interactúan dentro del sistema:

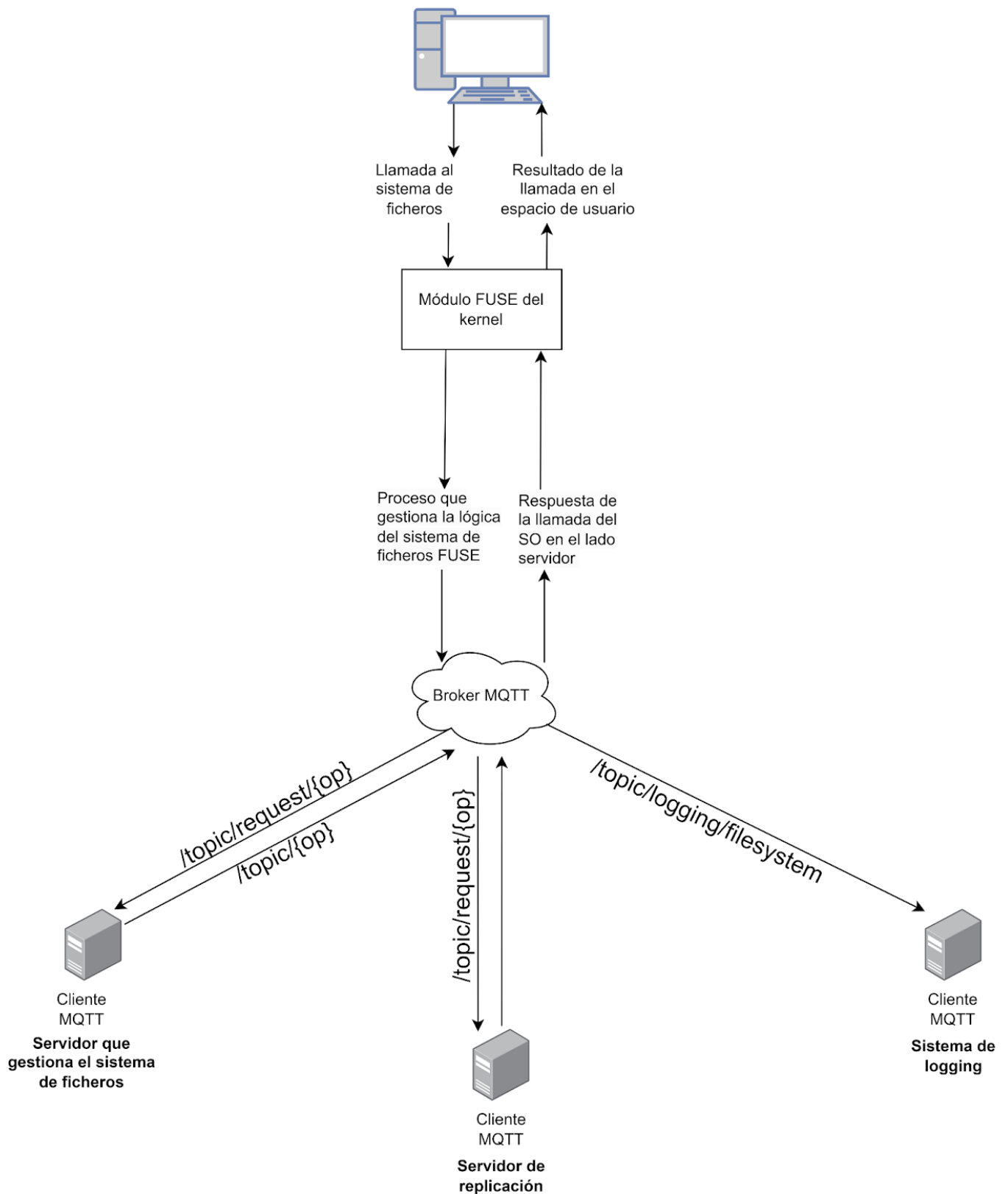


Figura 9. Arquitectura de la solución

Al utilizar el protocolo MQTT para la implementación de nuestro sistema de ficheros distribuido, los dos elementos más importantes son los siguientes:

- El broker MQTT
- Los clientes MQTT

En el diagrama de arquitectura no se representa al cliente FUSE como un cliente MQTT, pero también lo es. Para poder comunicarse con el cliente MQTT que gestiona el almacenamiento de los archivos, es necesario que se produzca el intercambio de solicitudes a través del protocolo. Aunque todos los nodos se consideran clientes, a efectos prácticos se podría considerar que se está utilizando una arquitectura cliente-servidor para este proyecto:

- **El cliente:** la máquina local donde el usuario monta su sistema. Este sistema gestiona los ficheros de manera totalmente transparente gracias al módulo FUSE.
- **El servidor:** el cliente MQTT que recibe las peticiones y ejecuta las operaciones en un servidor Linux. Después de ejecutar las operaciones manda el resultado al cliente para que vea los cambios reflejados en su máquina.

El flujo del sistema es el siguiente:

1. El usuario realiza alguna operación dentro del punto de montaje.
2. Está llamada es interceptada por el módulo FUSE del kernel.
3. Después el módulo FUSE llama al proceso donde se ha implementado la lógica que queremos ejecutar para el sistema de ficheros (el código Python con la librería de fusepy).
4. La librería fusepy proporciona una interfaz con la implementación de las distintas llamadas (open, write, read, etc.). Dentro de estas funciones se encuentra la implementación de MQTT que permite la comunicación con el otro cliente.

El broker hace de intermediario entre los clientes MQTT, y manejan los siguientes tópicos:

- El cliente que implementa FUSE publicará en el tópico *“/topic/request/{op}”*. Donde *“{op}”* representa a modo de variable cada una de las operaciones que se han implementado.
- El servidor que gestiona las peticiones publica en el tópico *“/topic/{op}”*
- El servidor que funciona como nodo de replicación no tiene necesidad de publicar en ningún tópico. Se utilizará solo para asegurar que los cambios de escritura mandados por el cliente tienen una copia de seguridad en otro nodo. En caso de que exista conflicto de versiones, se accederá y se hará un control manual de los datos
- El cliente que gestiona los ficheros de logs tampoco tiene necesidad de publicar en ningún tópico, solo escucha el resultado de las operaciones para poder almacenar el resultado de dichos eventos.

Por otro lado, las suscripciones a los tópicos serán las siguientes:

- El cliente que implementa FUSE está suscrito al topic *“topic/{op}”*

- El cliente MQTT que gestiona las peticiones está suscrito al tópico *“topic/request/{op}”*
- El cliente MQTT de replicación también está suscrito a *“topic/request/{op}”* para escuchar las peticiones y poder replicarlas
- El cliente MQTT que gestiona los logs escucha en el tópico *“topic/logging/filesystem”*

El Broker MQTT recibe la respuesta del cliente MQTT que actúa en el lado servidor, le manda la respuesta al lado cliente, y este la redirige internamente al módulo del kernel. Finalmente se ve el resultado reflejado en el espacio de usuario.

3.3.2 Diagrama de clases

Se ha decidido diseñar un diagrama de clases para tener una visión general de los distintos componentes antes de proceder a codificarlos.

El código se va a desarrollar en tantos ficheros de Python como clientes MQTT haya. Como en principio se quieren desarrollar los siguientes clientes:

- El cliente MQTT que realiza la manipulación de ficheros a través de FUSE.
- El cliente MQTT que gestiona las peticiones.
- El cliente MQTT que gestiona los logs.
- El cliente MQTT que va a trabajar en la replicación de los datos.

Entonces la solución estará compuesta de cuatro ficheros Python, con los siguientes nombres:

- **fuse_client.py** : el cliente que hace uso de la interfaz FUSE.
- **file_manager.py** : gestiona las operaciones en el lado servidor.
- **logging_mqtt.py** : para gestionar los logs.
- **file_replicator.py** : servidor de replicación.

A continuación se va explicar el paradigma de programación que se usará en cada uno de ellos:

- **Código que publica las peticiones para la ejecución de operaciones (fuse_client.py)**

Este código va a hacer uso de la librería “fusepy” como se ha mencionado anteriormente. Para ello se va a definir un método main() en el que se va a crear una instancia de la clase “FUSE”, y a la que se le va a pasar por argumento la clase “MqttFS()” donde están definidas todas las operaciones que se han implementado del sistema de ficheros. Como segundo argumento se pasa el directorio donde se va a montar el sistema de ficheros.

La clase “MqttFS” hereda de la clase “Operations” de “fusepy”, donde se sobrescriben todas las implementaciones de las funciones. El constructor va a tener los siguientes atributos:

- **pending_requests**: diccionario que ayuda a estructurar las peticiones.
- **mqtt.Client**: el objeto cliente MQTT

Por otro lado las siguientes funciones se ha decidido definirlas de forma global:

- **no_response_handler()**: maneja las operaciones de ficheros que no tienen respuesta.
- **response_handler()**: maneja las operaciones de ficheros con respuesta

El diccionario “pending_requests” se pasa como argumento al cliente MQTT a través del parámetro “userdata”, y se va a utilizar para manejar la asincronía como se explicará más adelante en el capítulo de implementación.

En este script por tanto hay parte de programación orientada a objetos, con la creación de la instancia FUSE y la definición de la clase “MqttFS” con sus respectivos métodos y atributos, y por otro lado hay parte del código que sigue un enfoque de programación procedimental, que es toda la parte relacionada con el desarrollo de MQTT.

Se ha seguido este enfoque porque:

- Al crear la instancia de la clase FUSE es necesario pasar como argumento una instancia de la clase que hereda de “Operations”, en este caso se ha llamado “MqttFS”
- Por otro lado puede ser conveniente separar la implementación de FUSE y MQTT, así como las funciones que garantizan su sincronía (sync(), response_handler()/no_response_handler()) .

Esta combinación de paradigmas de Programación Orientada a Objetos y Programación Procedimental permite un código más modularizado y legible.

- **El resto de códigos**

Se hará uso de los *callbacks* MQTT y la lógica se implementa dentro de la función “on_message”. Es por este motivo que se usa un paradigma de programación procedimental. No se estructurarán las funciones dentro de ninguna clase, se definen globalmente.

Aunque un diagrama de clases UML se suele diseñar para proyectos de programación orientada a objetos, también es útil para representar las distintas variables y funciones que se van a utilizar dentro de la programación procedimental, y para representar los atributos y métodos de las clases en caso de que se implementen.

Por lo que el siguiente esquema servirá a modo de guía para la etapa de implementación:

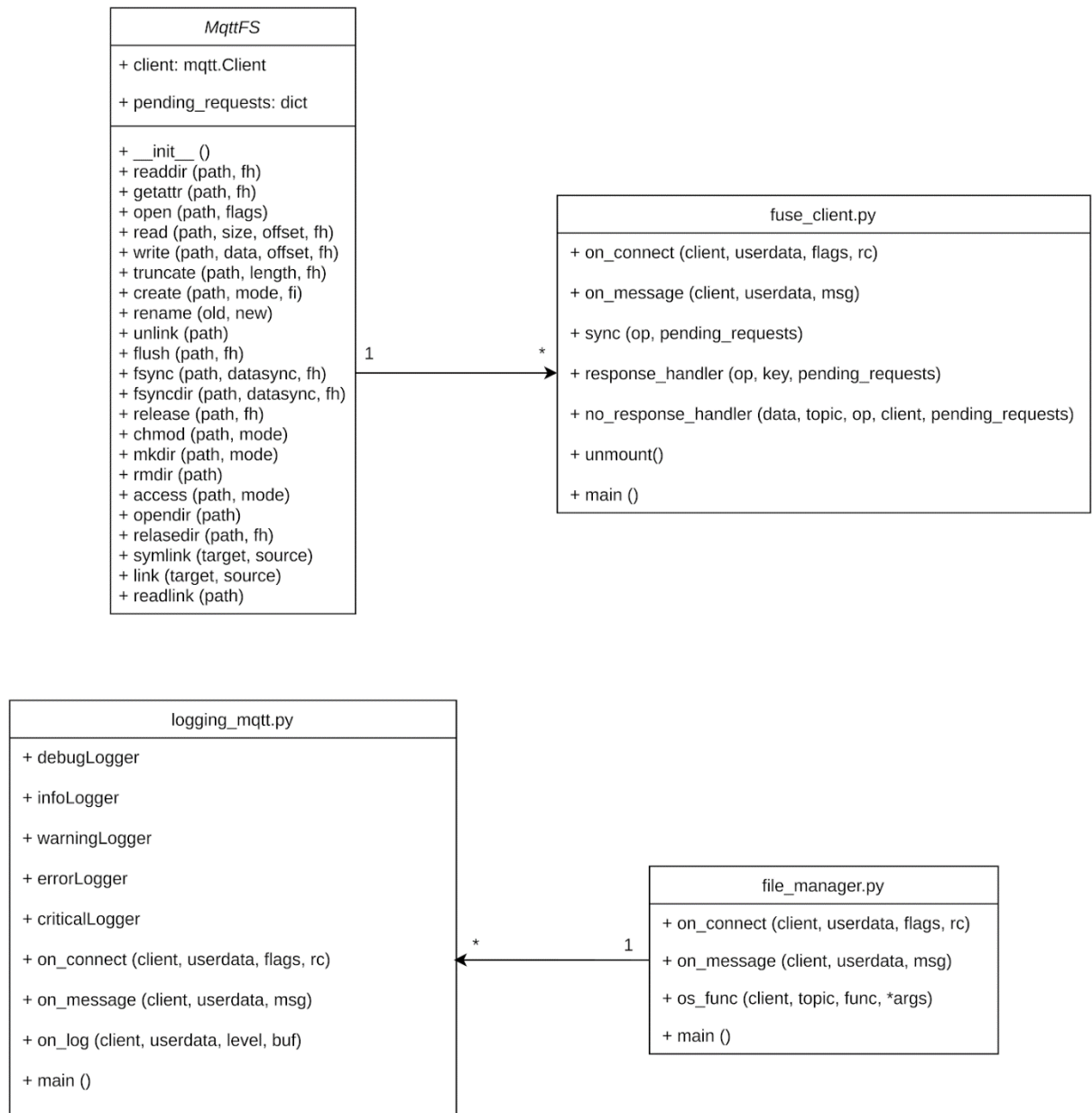


Figura 10. Diagrama de clases

Aunque no sigue la convención formal que suelen seguir los diagramas de clases UML, el principal objetivo del esquema era definir los nombres de:

- Clases
- Funciones
- Atributos
- Instancias

Como no existe el encapsulamiento como tal en Python, ya que un atributo o función se considera privado solo cuando sigue una convención de nombre (el nombre comienza con doble guión), se han puesto todos los atributos y funciones como públicos, por ello se ha utilizado la notación “+”.

3.3.2.1 Cardinalidad

A pesar de que no todos los elementos son clases, se ha indicado su relación para mejorar la comprensión del sistema. A continuación se explica la razón por la que se define la cardinalidad que aparece en el esquema:

- Para la relación “MqttFS” → “fuse_client.py” se ha puesto cardinalidad 1→* porque la instancia de la clase llama numerosas veces al resto de funciones globales.
- Para la relación “file_manager” → “logging_mqtt.py” se ha puesto cardinalidad 1→* porque todas las operaciones que se registran en los ficheros log propios del módulo “logging_mqtt.py” se disparan mediante el *callback* “on_message” que es llamado por el *publish* de “file_manager.py”.

3.3.2.2 Funciones FUSE

En esta tabla se presentan todas las funcionalidades que se han implementado:

Función	Descripción
open	Operación que sirve para abrir ficheros regulares.
read	Operación que sirve para leer ficheros regulares.
write	Operación que sirve para escribir ficheros regulares.
truncate	Operación que sirve para truncar un fichero (es decir, cambiar el tamaño del mismo).
create	Operación que sirve para crear ficheros regulares.
rename	Operación que sirve para renombrar ficheros.
release	Operación que sirve para cerrar ficheros y liberar los recursos asociados.
flush	Operación que realiza tareas de limpieza asociadas al cierre del archivo.
fsync	Operación que se encarga de sincronizar los cambios pendientes del fichero con el almacenamiento persistente.
getattr	Operación que recupera los atributos de un fichero (permisos de acceso, tamaño...).
access	Operación que verifica los permisos de un fichero antes de su manipulación.
opendir	Operación que sirve para abrir directorios.
readdir	Operación que sirve para leer directorios.
mkdir	Operación que sirve para crear directorios.

rmdir	Operación que sirve para borrar directorios.
releasedir	Operación que sirve para cerrar directorios.
fsyncdir	Misma implementación que “ <i>fsync</i> ”.
link	Operación que sirve para crear enlaces “duros”.
symlink	Operación que sirve para crear enlaces “simbólicos”.
readlink	Operación que sirve para leer un enlace “simbólico”.
unlink	Operación que sirve para eliminar enlaces “simbólicos”.
chmod	Operación que sirve para cambiar los permisos de acceso “ <i>rwX</i> ” (lectura, escritura y ejecución) para un fichero: Se pueden cambiar los permisos para el creador del fichero, para el grupo al que pertenece, y para “otros”.

Tabla 52. Funciones de fusepy

3.3.3 Diagramas de secuencia

Los diagramas de secuencia forman parte de la categoría de “Diagramas de interacción” dentro del estándar UML. Permiten representar la secuencia de eventos a lo largo del tiempo, entre el usuario, y los distintos componentes que forman parte del sistema. Esto es útil para modelar este proyecto, puesto que MQTT está diseñado para responder a eventos determinados, y de este modo se puede observar fácilmente el flujo entre mensajes.

3.3.3.1 Diagrama de secuencia de operación con respuesta

El siguiente diagrama representa el flujo de mensajes MQTT cuando la operación FUSE necesita retornar una respuesta. Por ejemplo, cuando se hace uso de la operación “open”, es necesario devolver el descriptor de fichero.

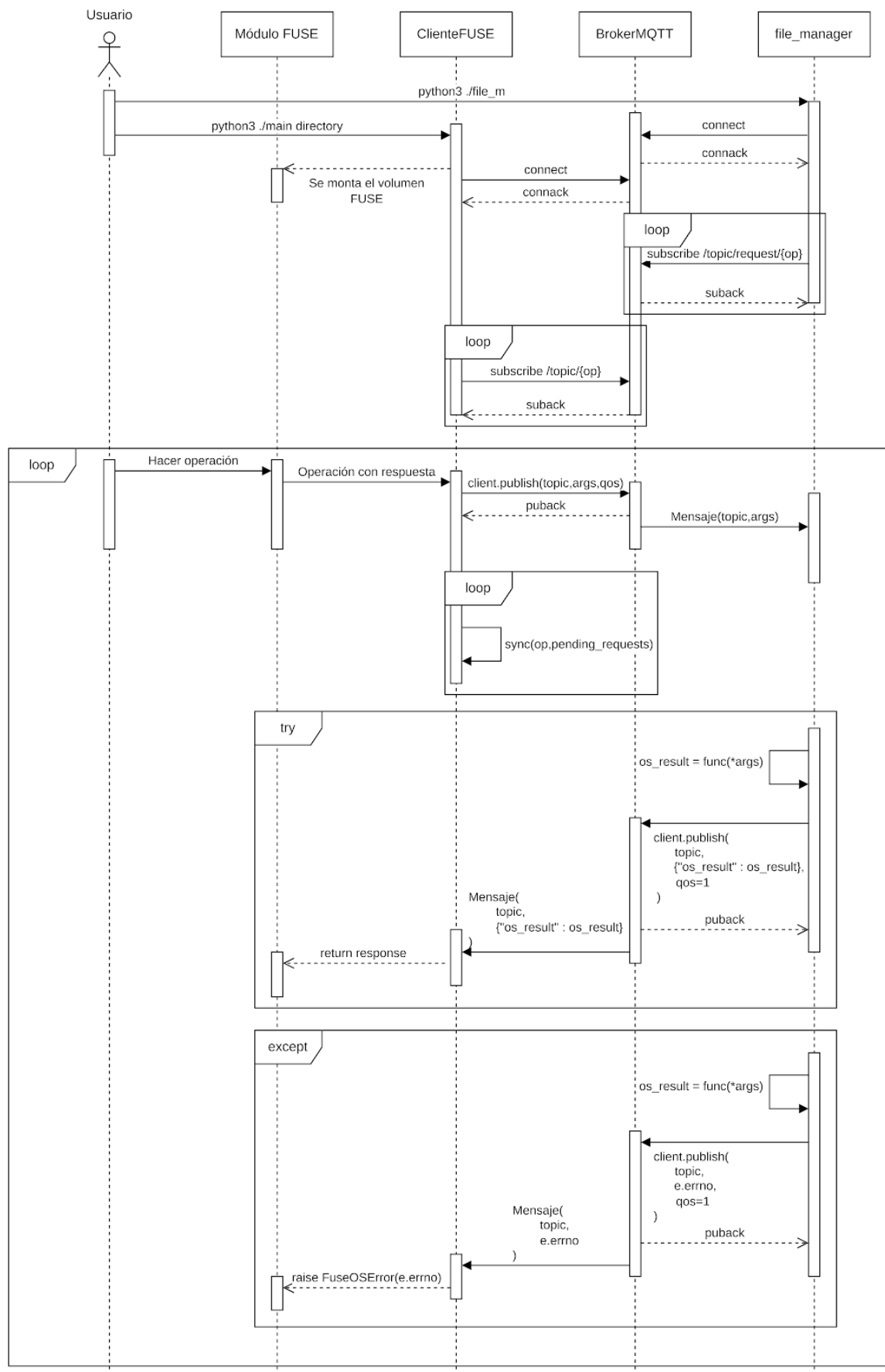


Figura 11. Diagrama de secuencia I

A continuación se va a explicar brevemente los aspectos clave del diagrama:

Por un lado, se puede observar que hay una función “sync(op,pending_requests)” dentro de un componente “loop”. El propósito de esta función es proporcionar un mecanismo de sincronización entre los clientes MQTT. El cliente FUSE solicita la respuesta de una determinada operación, y se hace espera activa dentro de “sync()” para que no siga el flujo de ejecución hasta que no se obtenga la respuesta por parte del “file_manager”.

Después nos encontramos ante dos posibles caminos que puede seguir la ejecución del sistema.

- **En caso de éxito**, se ejecutará la operación solicitada y se publicará el resultado de dicha operación, almacenándose en una estructura de tipo diccionario cuya clave será “os_result”.
- **En caso de error** se publicará el código de error mediante la creación de una instancia llamada “e” de la clase OSError, y se publicará el atributo “errno”, que contiene dicho código identificativo.

Para el caso de éxito se publica el resultado de la operación en una estructura de tipo diccionario, para poder hacer una diferenciación con el resultado de error mediante la comparación de tipos.

Para simplificar:

- **Si el cliente FUSE recibe un diccionario en el callback on_message**
Se trata de la respuesta de la operación que hemos solicitado y la cual no ha producido errores
- **Si el cliente FUSE recibe un integer con valor distinto de 0 en el callback on_message**
Se trata de un código de error y se levantará una excepción mediante la clase “FuseOSError”

Después en el cliente FUSE se redirige la respuesta a la función “response_handler”.

3.3.3.2 Diagrama de secuencia de operación sin respuesta

El siguiente diagrama representa el intercambio de mensajes MQTT cuando la operación FUSE no requiere una respuesta, es decir retorna *None*.

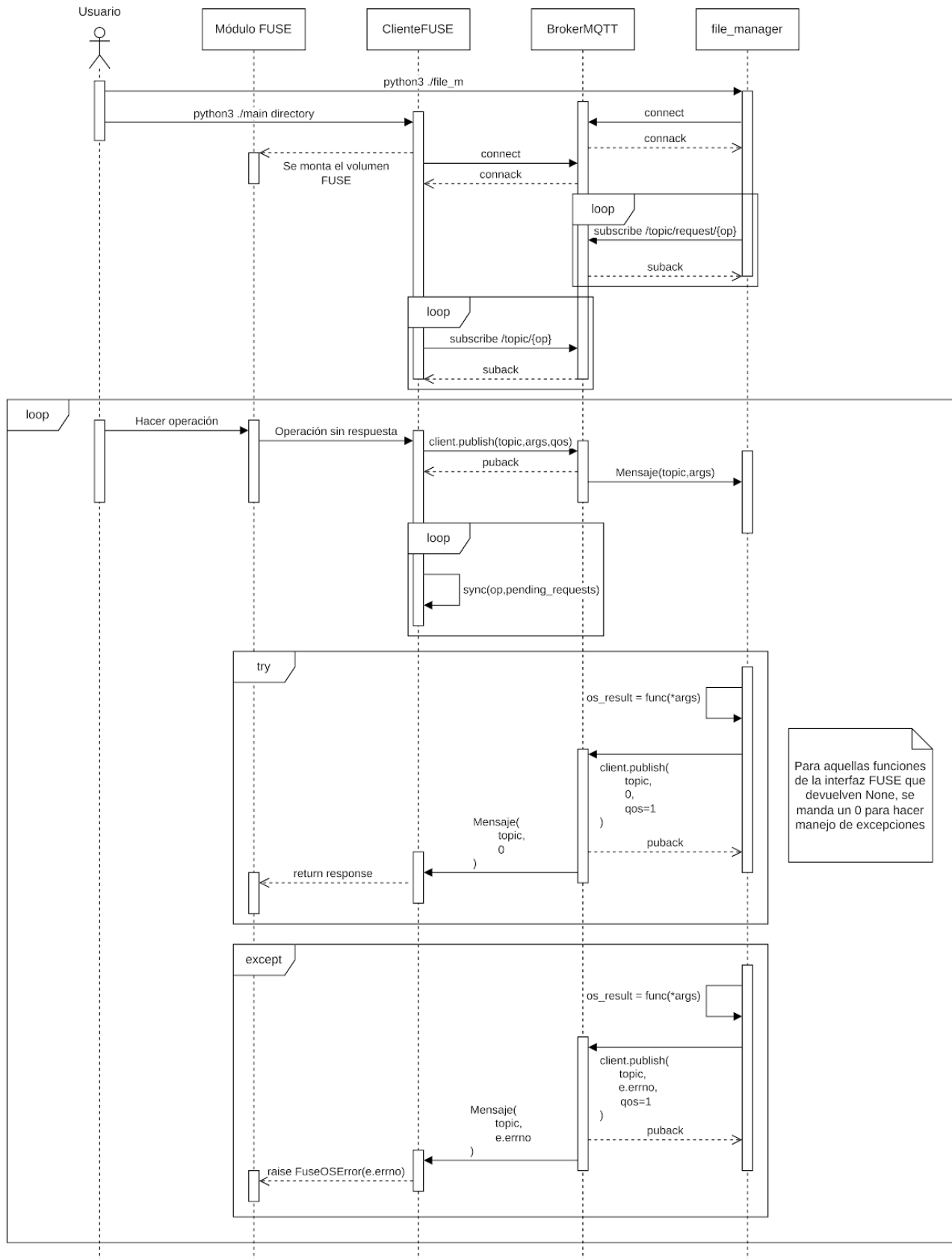


Figura 12. Diagrama de secuencia II

En caso de éxito simplemente se envía un 0 para diferenciar la respuesta con la recibida en caso de error.

Después el cliente FUSE redirige la respuesta a la función “no_response_handler”.

3.3.3.3 Diagrama de secuencia del sistema de logging

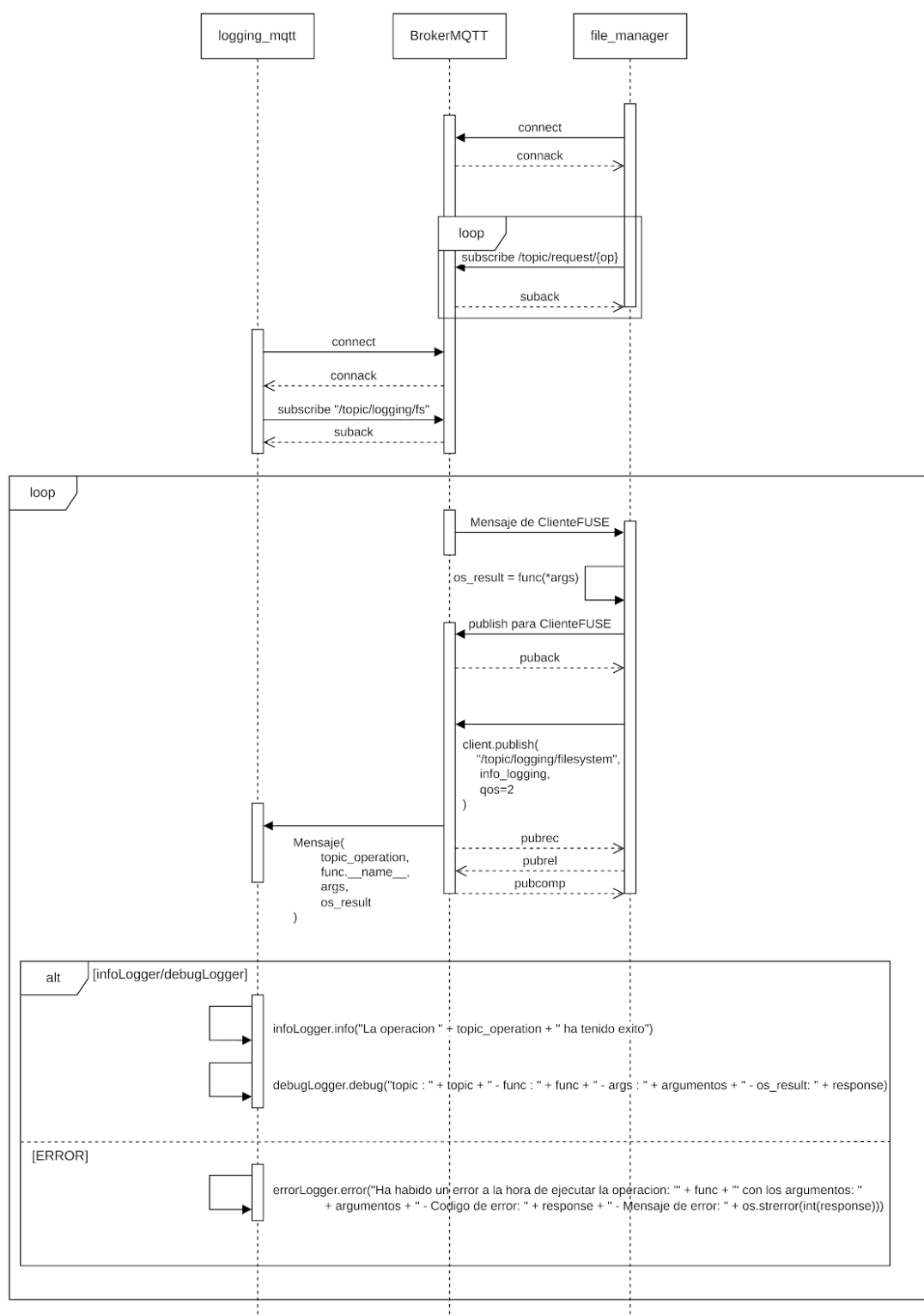


Figura 13. Diagrama de secuencia III

En este diagrama se ha simplificado toda la parte relacionada con FUSE para ofrecer una mayor claridad. El sistema de *logging* recibe los mensajes del “file_manager”, el cual publica la siguiente información adicional al tópico del *logger*:

- El nombre de la función
- Los argumentos que se han utilizado
- El nombre del tópico

Esto es capturado por el cliente MQTT que realiza las gestiones relacionadas con el *logging*, y captura los logs en estos tres posibles niveles:

- INFO
- DEBUG
- ERROR

3.4 Implementación

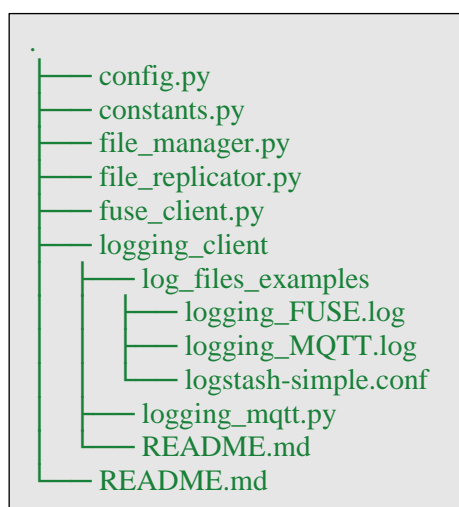
Durante esta fase se codifica la solución de software en base a las etapas de análisis y diseño. Se van a documentar los aspectos técnicos que han supuesto una mayor dificultad, y se va a explicar de forma genérica el código, explicando cuál es el propósito detrás de cada función. También se va a explicar cómo se organiza la estructura del proyecto y cuál es el propósito de cada fichero dentro del repositorio.

El código está subido a un repositorio de GitHub y se puede ver desde el siguiente enlace:

https://github.com/DiegoUC3M/mqtt_file_system_tfg/

3.4.1 Estructura del proyecto

Como se puede observar en el repositorio, el proyecto sigue la siguiente estructura:



A continuación se explica brevemente el propósito de cada archivo:

- **fuse_client.py**

Cliente MQTT que utiliza la interfaz FUSE.

- **file_manager.py**

Cliente MQTT que se encarga de la gestión de los ficheros, y que actúa como “lado servidor” dentro del sistema de ficheros distribuido.

- **file_replicator.py**

Cliente MQTT que se puede instalar en un servidor distinto para proporcionar tolerancia a fallos mediante el uso de replicación.

- **config.py**

Archivo de configuración para establecer los siguientes parámetros para los clientes MQTT:

- MQTT_BROKER_HOST
- MQTT_BROKER_PORT
- MQTT_BROKER_KEEPALIVE
- MQTT_BROKER_USER
- MQTT_BROKER_PASSWORD

- **constants.py**

Fichero donde se almacenan todas las constantes. En este caso están almacenados los nombres de los tópicos que se van a utilizar para la comunicación MQTT.

- **logging_client**

Directorio que cuenta con los siguientes ficheros:

- **logging_mqtt.py**

Este cliente MQTT se encarga de la monitorización y registro de eventos.

- **log_files_examples**

- **logging_FUSE.log**

Fichero log donde se muestra cómo se almacenan los registros de las operaciones relacionadas con el sistema de ficheros.

- **logging_MQTT.log**

Fichero log donde se almacenan los eventos relacionados con la comunicación MQTT.

- **logstash-simple.conf**

Archivo de configuración de Logstash que sirve para gestionar los archivos de log creados con “logging_mqtt.py”, y que permite el

almacenamiento en Elasticsearch, y la visualización y análisis de registros en Kibana.

3.4.2 Repaso de las distintas funciones existentes dentro del código

Se va a explicar el flujo del programa y las funciones asociadas. Primero, el cliente FUSE envía una petición. Después, se ejecuta la función “os_func” en “file_manager.py”, y este envía la respuesta al *handler* correspondiente en “fuse_client.py”, donde se devuelve la respuesta o se levanta una excepción al módulo FUSE. En caso de que no tenga respuesta se manda a “no_response_handler” y en caso de que sí la tenga se envía a “response_handler”.

3.4.2.1 fuse_client.py

- **sync (op, pending_requests)**

Esta función sincroniza la respuesta de “file_manager.py” con el manejo de la petición de “fuse_client.py”.

Cuando el cliente publica la petición, es necesario esperar para obtener la respuesta por parte del “lado servidor”. Cuando se sincronice la respuesta y se pueda retornar el resultado de la operación a través de FUSE, el efecto se hará visible en el sistema de ficheros. Para ello se hace uso de la espera activa, y tiene como parámetros:

```
def sync(op, pending_requests):  
    while pending_requests.get(op) is None:  
        time.sleep(0.1)  
    return pending_requests.pop(op)
```

Figura 14. Función sync

- **op**: la operación de la que se espera obtener el resultado.
- **pending_requests**: diccionario en el cual se va a almacenar el resultado de la operación y cuya clave es el nombre de la operación, y su valor el resultado. Este diccionario se pasa por referencia a “userdata” a la hora de crear el cliente MQTT, y cuando llega un mensaje, el evento es capturado por el *callback* “on_message” guardando dicho resultado en “userdata” (que como apunta a la misma referencia en memoria, es accesible desde “pending_requests”).

```
# Para manejar la asincronia:  
self.pending_requests = {}  
  
self.client = mqtt.Client(client_id="main", userdata=self.pending_requests)
```

Figura 15. Creación del cliente MQTT con userdata

El resultado de las operaciones es capturado así por “on_message”:

```
def on_message(client, userdata, msg):
    topic = msg.topic.split('/')
    userdata[topic[-1]] = msg.payload.decode()
```

Figura 16. Callback on_message del cliente FUSE

- **response_handler(op, key, pending_requests)**
Es el “manejador” al que llaman las operaciones que devuelven algún tipo de respuesta. El propósito de esta función es el de modularizar el código para que sea reutilizable y legible.
En caso de error, se levanta la excepción correspondiente.
- **no_response_handler(data, topic, op, client, pending_requests)**
Es el “manejador” al que llaman las operaciones que no devuelven ninguna respuesta. Para poder hacer el manejo de excepciones y saber si es necesario levantar una excepción, se compara con un 0 cuando la operación ha sido exitosa, y con cualquier Integer mayor que 0 en el caso contrario.

3.4.2.2 file_manager.py

- **os_func(client, topic, func, *args)**

Sirve para modularizar el código y separar funcionalidades. En el “on_message” se produce la parte de capturar el mensaje y deserializar los argumentos, y después se llama a esta función para realizar la llamada correspondiente.

3.4.3 Dificultades encontradas en la etapa de implementación

Algunas funciones cuentan con una implementación un poco más compleja. En este apartado se profundiza en el desarrollo de las operaciones de lectura y escritura.

Operación write/read

Como se puede observar en el código, es necesaria una codificación en base64 cuando se necesita enviar el texto de algunos ficheros a través de MQTT.

Por ejemplo, cuando el usuario genera algún fichero de texto, también se generan ficheros “.swp”. Estos ficheros “.swp” almacenan información temporal, como su nombre bien indica (ficheros swap). Si se observan los logs generados:

[illegible]

Figura 17. Registros log

Se puede observar que se abre el fichero “.e.swp” con descriptor de fichero “4”, y después una operación *write* escribe la cadena de bytes que aparece en la figura, sobre ese descriptor de fichero.

Al ver la implementación del *write* dentro de MqttFS, se puede observar que es necesario pasar los argumentos que se van a procesar dentro de *os.write()*. Para ello los argumentos se deben almacenar en un diccionario que después se va a serializar con la librería JSON. En este caso “data” es una cadena de bytes, y se quiere asignar ese valor a la clave “write_data[“text”]”.

```
def write(self, path, data, offset, fh):
    write_data = {}
    write_data["file_handle"] = fh
    write_data["text"] = base64.b64encode(data).decode()

    datos_json = json.dumps(write_data)
    self.client.publish(REQUEST_WRITE_TOPIC, datos_json,
                        qos=2) # Solicito escritura.
```

Figura 18. Implementación de la función de escritura

El problema radica en que ninguno de los valores del diccionario puede ser de tipo binario, ya que la serialización con JSON no lo permite, y los ficheros “.swp” almacenan metadatos y datos que no sigue una codificación binaria estándar como utf-8 (es decir, no se puede transformar la cadena de bytes a texto mediante un simple “decode”).

Por lo que la solución pasa por codificar los datos a base64 y después transformar la cadena de bytes resultante a texto. Después en el lado servidor será necesario hacer la decodificación en base64 correspondiente.

Este procedimiento es necesario también para las operaciones de lectura por la misma razón.

3.5 Implantación

Se van a describir los pasos necesarios para el despliegue de la solución. Es indispensable disponer de dos máquinas con Linux. También es necesario disponer de conexión a internet para que se pueda producir la comunicación MQTT.

En su defecto, se podría ejecutar todo en una única máquina (incluido el broker) de forma local. De este modo se puede estudiar cómo funciona el proyecto y comprobar su correcto funcionamiento.

3.5.1 Instalación de dependencias

En caso de no tener instalado Python:

```
sudo apt update
sudo apt install python3
```

Primero se va a proceder con la instalación de las siguientes dependencias:

```
pip install fusepy  
pip install "paho-mqtt<2.0.0"
```

Estos paquetes no vienen por defecto con Python y es necesario instalarlos. El código se ha realizado con una versión de “paho-mqtt” anterior a la actual, ya que esta última versión salió a principios del año 2024, cuando el proyecto ya estaba en desarrollo.

Para poder utilizar Mosquitto es necesario instalar el broker en alguna de las máquinas, para ello basta con ejecutar el siguiente comando, en el caso de encontrarnos con un sistema basado en Debian, como puede ser Ubuntu:

```
sudo apt install mosquitto mosquitto-clients
```

3.5.2 Arranque y configuración

Primero es necesario tener un broker MQTT al cuál se puedan conectar los clientes. Una vez elegido el Broker MQTT, es necesario establecer el host, el puerto, y el keepalive en el fichero “config.py”. Por ejemplo si se ejecuta el broker localmente:

```
MQTT_BROKER_HOST = "localhost"  
MQTT_BROKER_PORT = 1883  
MQTT_BROKER_KEEPALIVE = 60  
MQTT_BROKER_USER = "user"  
MQTT_BROKER_PASSWORD = "password"
```

Los parámetros anteriores son utilizados por la biblioteca “paho-mqtt” y son utilizados independientemente de la implementación del broker MQTT que se haya elegido, por lo que este paso es común sea cual sea la elección. Las variables son recuperadas por los clientes MQTT para realizar la conexión con el broker.

3.5.3 Configuración con Mosquitto

La configuración de un usuario y una contraseña no son comunes para todos los Brokers, por lo que voy a explicar el procedimiento si se quiere usar Mosquitto, que es el Broker que se ha utilizado para las pruebas durante la codificación del proyecto. Cabe mencionar que es necesario tener privilegios de superusuario para poder realizar este procedimiento.

En caso de no querer permitir conexiones anónimas, se puede realizar la siguiente configuración en Mosquitto. Primero es necesario crear el archivo que va a contener las contraseñas:

```
sudo touch /etc/mosquitto/passwd
```

Después creamos un usuario y contraseña, introduciendo cualquier valor que desee el usuario. Estos valores tienen que coincidir con los valores establecidos en el fichero “config.py”:

```
sudo mosquitto_passwd -b /etc/mosquitto/passwd usuario_aqui contraseña_aqui
```

Hay que modificar el archivo de configuración para que haga uso del archivo de contraseñas. Para ello se va a crear un fichero de configuración en el directorio “/etc/mosquitto/conf.d”.

```
sudo nano /etc/mosquitto/conf.d/mosquitto.conf
```

Y añadir las siguientes líneas:

```
listener 1883  
  
allow_anonymous false  
  
password_file /etc/mosquitto/passwd
```

En caso de no querer permitir el uso de credenciales ni conexiones anónimas basta con no seguir ninguno de los pasos anteriores.

Si se desea que Mosquitto escuche en un puerto y dirección IP distintos se puede modificar la siguiente línea del archivo de configuración:

```
listener inserte_el_puerto_aqui inserte_ip_aqui
```

Si Mosquitto se está iniciando automáticamente con “systemd” y no se desea que siga este comportamiento, se puede deshabilitar e iniciarlo manualmente cuando se desee.

Para deshabilitarlo:

```
sudo systemctl disable mosquitto
```

Para arrancar el servicio Mosquitto manualmente:

```
mosquitto -v -c /etc/mosquitto/conf.d/mosquitto.conf
```

Se puede añadir el parámetro -v para ver el output en modo “verbose”. Con la opción -c se especifica el archivo de configuración que debe utilizar Mosquitto para poder desplegarse.

3.5.4 Despliegue del sistema

Es necesario ejecutar el código del lado cliente junto con un argumento, donde se define la ubicación del sistema de ficheros. El directorio puede introducirse como una variable global dentro del fichero Python, pero se recomienda hacerlo del siguiente modo:

```
python3 ./fuse_client.py "/path/del/directorio/donde/deseas/montar/tu/FS/"
```

El resto de códigos se ejecutan de la misma manera.

La elección de un Broker MQTT en la nube que maneje una cantidad tan elevada de mensajes, y la elección de un servidor que aloje los ejecutables, supone un coste elevado. Por ello se ha decidido implementar y realizar las pruebas en local, pero la solución es totalmente adaptable a cualquier Broker y servidor con Linux.

Una de las soluciones más económicas a largo plazo pasa por ejecutar Mosquitto o el resto de ficheros en una o más Raspberry Pi’s, para no tener que depender de servicios de terceros.

3.5.5 Implantación de ELK

Desplegar ELK en la nube puede llegar a ser costoso, pero si se instala en local se puede utilizar gratuitamente, al ser una solución de código abierto.

Simplemente es necesario descargar las carpetas con los ejecutables desde la página de Elastic, y realizar los siguientes pasos:

Dentro de la carpeta de Elasticsearch hay que ejecutar:

```
bin/elasticsearch
```

Durante la primera ejecución de Elasticsearch, se genera una contraseña que permite iniciar sesión dentro de Kibana. Esta contraseña aparece por la terminal y es conveniente apuntarla.

Después, dentro de la carpeta de Kibana es necesario ejecutar:

```
bin/kibana
```

Y dentro de la carpeta de Logstash:

```
bin/logstash -f logstash-simple.conf
```

Con la opción “-f” se indica la ubicación del archivo de configuración de Logstash. En este caso se llama “logstash-simple.conf”.

En este fichero se indica qué transformación deben seguir los mensajes dentro de los ficheros log para poder ser importados a Elasticsearch.

Una vez hecho todo esto, podemos comprobar que el servicio de Elasticsearch funciona correctamente al conectarse al puerto 9200:

`https://localhost:9200/`

Pedirá usuario y contraseña. En el caso de no haber modificado ninguna de las dos:

- **Usuario:** elastic
- **Contraseña:** la que se ha generado durante la ejecución de Elasticsearch

Para poder utilizar Kibana es necesario introducir las mismas credenciales, esta vez en el puerto 5601:

`http://localhost:5601/`

4. EVALUACIÓN

Para comprobar el correcto funcionamiento de la solución, verificar que se hayan implementado todas las funcionalidades que se han propuesto, y evitar que aparezcan errores inesperados, es necesario realizar una evaluación del sistema. En este caso se ha diseñado un plan de pruebas.

4.1 Diseño del plan de pruebas

El plan de pruebas garantiza el correcto funcionamiento de los distintos módulos que componen el proyecto, y verifica que se cumplen los distintos requisitos que se marcaron en la fase de análisis. Gracias a este plan de pruebas podemos realizar un seguimiento más exhaustivo que garantiza la calidad del software desarrollado.

Se va a establecer la siguiente plantilla de tipo tabla donde se van a definir todas las pruebas que se van a verificar.

P-XX	
Descripción	
Función	
Entrada	
Salida	
Requisitos verificados	

Tabla 53. Plantilla de casos de prueba

Se va a seguir la siguiente notación:

- **Identificador de prueba:** Este campo es un “ID” que localiza unívocamente cada prueba. Se va a seguir la siguiente convención:
 - “P-XX”
 - “P”: Prueba
 - “XX”: Número que identifica cada prueba, en orden de aparición
- **Descripción:** Breve descripción indicando cuál es el propósito de la prueba, y qué es lo que se quiere verificar.
- **Función:** Indica para qué función del código se está realizando la prueba.
- **Entrada:** Los datos o parámetros que se han de proporcionar para que sea posible obtener la salida deseada.
- **Salida:** Indica cuál es el resultado que se desea obtener. Se habrá verificado correctamente la prueba si el resultado obtenido y el deseado coinciden.

- **Requisitos verificados:** En este campo se muestran los requisitos que están relacionados con este caso de prueba y que han sido verificados a través del mismo.

4.2 Casos de prueba

Se ha intentado reducir el número de casos de prueba debido a la similitud entre algunos de ellos. Poniendo de ejemplo el primer caso de prueba, es necesario definir “opendir(path)” para seguir la interfaz definida por FUSE, pero internamente llama a la función “open()” de la biblioteca “os”.

Los siguientes casos de prueba están estrechamente relacionados con los requisitos, ya que es imprescindible verificar al menos todo aquel que tenga un nivel de verificabilidad alto. Se expresa cuál ha de ser la salida deseada para que puedan ser verificados.

P-01	
Descripción	Apertura de ficheros y de directorios.
Función	open(path, flags) y opendir(path). Internamente llaman a os.open.
Entrada	Ubicación relativa del fichero que se quiere abrir junto con sus “flags”.
Salida	La salida de las funciones “open/opendir” es el descriptor de fichero y el usuario observa que el fichero se ha abierto.
Requisitos verificados	RF-04, RF-13

Tabla 54. P-01

P-02	
Descripción	Cierre de ficheros y directorios
Función	release(path, fh) y releasedir(path, fh). Ambas llaman a os.close(fh).
Entrada	Descriptor de fichero que se quiere cerrar.
Salida	El fichero/directorio se ha cerrado. Las funciones release/releasedir no devuelven nada.
Requisitos verificados	RF-07

Tabla 55. P-02

P-03	
Descripción	Escritura de ficheros.
Función	write(path, data, offset, fh). Internamente llama a os.write(fh, data).
Entrada	El descriptor de fichero, el desplazamiento y el texto que se va a escribir.
Salida	Devuelve el número de bytes que se han escrito en el fichero. El usuario verá los cambios de escritura reflejados en el mismo.
Requisitos verificados	RF-06

Tabla 56. P-03

P-04	
Descripción	Lectura de ficheros
Función	read(path, size, offset, fh). Internamente llama a os.read(fh, size).
Entrada	El descriptor de fichero y el número de bytes que se quieren leer.
Salida	Devuelve la cadena de texto solicitada.
Requisitos verificados	RF-05

Tabla 57. P-04

P-05	
Descripción	Obtener los atributos de ficheros y directorios.
Función	getattr(path, fh=None). Llama a os.lstat(path).
Entrada	La ubicación del fichero.
Salida	Diccionario con todos los atributos del fichero. Si un usuario hace un “ls -l” podrá ver los atributos de todos los ficheros y directorios.
Requisitos verificados	RF-18

Tabla 58. P-05

P-06	
Descripción	Leer las entradas de un directorio.
Función	readdir(path, fh). Llama a os.listdir(path).
Entrada	Directorio del cual se quieren leer las entradas.
Salida	Lista de strings con todas las entradas. Si un usuario hace un “ls” podrá ver el contenido del directorio.
Requisitos verificados	RF-17

Tabla 59. P-06

P-07	
Descripción	Crear un fichero.
Función	create(path, mode, fi=None). Llama a: os.open(path, os.O_WRONLY os.O_CREAT, mode)
Entrada	La ubicación donde se quiere crear el fichero, junto con sus permisos.
Salida	El descriptor de fichero. El usuario ve que el fichero se ha creado.
Requisitos verificados	RF-09

Tabla 60. P-07

P-08	
Descripción	Modificar el tamaño de un archivo.
Función	truncate(path, length, fh=None). Llama a os.truncate(path,length).
Entrada	La ubicación del fichero que se quiere truncar, y el tamaño nuevo que se desea establecer.
Salida	El usuario ve que el tamaño del fichero ha cambiado. La función no devuelve nada.
Requisitos verificados	

Tabla 61. P-08

P-09	
Descripción	Eliminar ficheros.
Función	unlink(path). Llama a os.unlink(path).
Entrada	El directorio del fichero que se quiere eliminar.
Salida	El fichero se ha eliminado. La función unlink no devuelve nada.
Requisitos verificados	RF-10, RF-22

Tabla 62. P-09

P-10	
Descripción	Renombrar ficheros y directorios.
Función	rename(old, new). Llama a os.rename.
Entrada	La ruta antigua del fichero (el nombre antiguo), junto con el nuevo.
Salida	El usuario puede ver que el fichero ha cambiado de nombre. La función rename no devuelve nada.
Requisitos verificados	RF-08. RF-14

Tabla 63. P-10

P-11	
Descripción	Cambiar los permisos de ficheros y directorios.
Función	chmod(path, mode). Llama a os.chmod.
Entrada	La ruta del fichero y los nuevos permisos.
Salida	El usuario puede ver con un “ls -lh” los nuevos permisos. La función chmod no devuelve nada.
Requisitos verificados	RF-11

Tabla 64. P-11

P-12	
Descripción	Creación de un directorio.
Función	mkdir(path, mode). Llama a os.mkdir.
Entrada	La ruta donde se quiere crear el directorio y los permisos asociados.
Salida	El usuario ve que se ha creado el nuevo directorio. La función mkdir no devuelve nada.
Requisitos verificados	RF-14

Tabla 65. P-12

P-13	
Descripción	Eliminar un directorio.
Función	rmdir(path). Llama a os.rmdir(path).
Entrada	La ruta del directorio que se quiere eliminar.
Salida	El directorio se ha eliminado. La función rmdir no devuelve nada.
Requisitos verificados	RF-15

Tabla 66. P-13

P-14	
Descripción	Verificar los permisos de un fichero.
Función	access(path, mode). Llama a os.access.
Entrada	La ruta del fichero y los permisos que se quieren comprobar.
Salida	Devuelve 0 (True) o -1 (False).
Requisitos verificados	RF-12

Tabla 67. P-14

P-15	
Descripción	Creación de un enlace duro o blando.
Función	symlink(target, source) / link(target, source). Llaman a os.symlink y os.link.
Entrada	La ruta del fichero del cual se quiere crear el enlace, y la ubicación del enlace.
Salida	El enlace contiene los mismos datos que el fichero al que se ha enlazado. Symlink y link no devuelven nada.
Requisitos verificados	RF-19, RF-20

Tabla 68. P-15

P-16	
Descripción	Devolver la ruta del fichero original al que apunta un enlace simbólico.
Función	readlink(path). Llama a os.readlink.
Entrada	La ruta del enlace.
Salida	La ruta del archivo original.
Requisitos verificados	RF-21

Tabla 69. P-16

P-17	
Descripción	El punto de montaje se puede introducir por parámetro. En caso de proporcionar más de dos argumentos no se ejecutará el programa.
Entrada	El path relativo del ejecutable + el path absoluto del punto de montaje: python3 /path/ejecutable /path/absoluto/punto/de/montaje
Salida	El directorio se ha montado correctamente en el espacio de usuario
Requisitos verificados	RF-01

Tabla 70. P-17

P-18	
Descripción	El usuario podrá utilizar una interfaz para interactuar con el sistema de ficheros, ya sea a través de la GUI o la terminal.
Entrada	
Salida	Interfaz accesible por el usuario.
Requisitos verificados	RF-03, RNF-04

Tabla 71. P-18

P-19	
Descripción	El sistema de ficheros utiliza MQTT para la comunicación.
Entrada	El broker MQTT tiene que estar iniciado.
Salida	Los clientes MQTT se han suscrito a los correspondientes tópicos y el broker es el intermediario en ese intercambio de mensajes.
Requisitos verificados	RF-02, RNF-03

Tabla 72. P-19

P-20	
Descripción	Registro de logs.
Entrada	Cualquier operación del sistema de ficheros.
Salida	Una entrada en el fichero “logging_MQTT.log” o “logging_FUSE.log” según corresponda, con su nivel correspondiente.
Requisitos verificados	RF-24, RNF-10

Tabla 73. P-20

P-21	
Descripción	Análisis y visualización de logs mediante ELK.
Entrada	Los ficheros log en Logstash.
Salida	Organización y visualización de datos en Kibana.
Requisitos verificados	RF-26, RNF-11

Tabla 74. P-22

P-22	
Descripción	Verificar la tolerancia a fallos.
Entrada	Tiene que estar montado el sistema de ficheros
Salida	Simulamos la caída del cliente que gestiona las peticiones, y vemos como el cliente de replicación sigue realizando las operaciones de escritura.
Requisitos verificados	RF-25

Tabla 75. P-22

5. VISUALIZACIÓN CON KIBANA Y PROCESAMIENTO CON LOGSTASH

Se va a explicar la configuración necesaria para poner en funcionamiento la pila ELK, se van a mostrar la organización y filtrado de los logs, y también se muestra la visualización de los datos a través de los *dashboards* y gráficos.

5.1 Archivo de configuración de Logstash

Para el procesamiento de los ficheros log en Logstash, ha sido necesario crear un archivo de configuración que procesa las entradas y transforma la información. Para ello es importante especificar la ubicación de los ficheros log, y su salida. En este caso se envían los datos a Elasticsearch, y la transformación se consigue a través de estos dos filtros:

- **grok**: Sirve para estructurar los mensajes de los ficheros log en distintos campos para su posterior manipulación.
- **date**: Se utiliza para la transformación de los datos de tipo “fecha”.

Dicho fichero de configuración se encuentra presente en el repositorio y se llama **logstash-simple.conf**.

5.2 Visualización de logs y dashboards

En el apartado Discover dentro de Kibana, nos encontramos con una pantalla principal donde se muestran las trazas de la ejecución del código de una forma estructurada, como se puede comprobar en la siguiente figura.

elastic

Find apps, content, and more.

Discover

New Open Share Alerts Inspect Save

Prueba

Filter your data using KQL syntax

Refresh

Search field names

7

Documents (7,557)

Field statistics

Columns 7

Sort fields

Selected fields

7

log_level timestamp topic func args os_result message

Popular fields

10

message mqtt_msg operation type args func log_level os_result timestamp topic

Available fields

19

@timestamp @version args error_code error_msg

Add a field

Get the best look at your search results

Take the tour Dismiss

log_level timestamp topic func args os_result message

DEBUG	2024-07-25 11:53:30	/topic/readDir	listdir	/home/diego/PycharmProjects/mqtt_test/directorio_servidor/	['.', '..', 'hola', '.que.swp', '.hola.swp', '.Trash-1000', 'j', 'neee']	DEBUG - 2024-07-25 11:53:30 - topic : /topic/readDir - func : listdir - args : ...
INFO	2024-07-25 11:53:32	-	-	-	-	INFO - 2024-07-25 11:53:32 - La operacion getattr ha tenido exito
DEBUG	2024-07-25 11:53:32	/topic/getattr	lstat	/home/diego/PycharmProjects/mqtt_test/directorio_servidor/neee	{'st_atime': 1721895109.8135474, 'st_ctime': ...}	DEBUG - 2024-07-25 11:53:32 - topic : /topic/getattr - func : lstat - args : ...
INFO	2024-07-25 11:53:33	-	-	-	-	INFO - 2024-07-25 11:53:33 - La operacion readDir ha tenido exito
INFO	2024-07-25 11:53:34	-	-	-	-	INFO - 2024-07-25 11:53:34 - La operacion getattr ha tenido exito
DEBUG	2024-07-25 11:53:34	/topic/getattr	lstat	/home/diego/PycharmProjects/mqtt_test/directorio_servidor/	{'st_atime': 1721817985.1425867, 'st_ctime': ...}	DEBUG - 2024-07-25 11:53:34 - topic : /topic/getattr - func : lstat - args : ...
INFO	2024-07-25 11:53:34	-	-	-	-	INFO - 2024-07-25 11:53:34 - La operacion getattr ha tenido exito
DEBUG	2024-07-25 11:53:36	/topic/open	open	/home/diego/PycharmProjects/mqtt_test/directorio_servidor/', 65536	4	DEBUG - 2024-07-25 11:53:36 - topic : /topic/open - func : open - args : ...
DEBUG	2024-07-25 11:53:37	/topic/release	close	4	-	DEBUG - 2024-07-25 11:53:37 - topic : /topic/release -

Rows per page: 100

1 2 3 4 5

Figura 19. Pantalla Discover de Kibana

Hay 7.557 registros en total en el momento en el que se tomó la captura. Este es el aspecto más importante de la pila ELK, ya que permite gestionar eficientemente los datos y buscar entre un número elevado de registros en base a un atributo concreto. Esta sería una tarea muy complicada sin la ayuda de un motor de búsqueda como el que nos ofrece Elasticsearch.

A la izquierda aparece una columna con todos los campos que podemos añadir para la visualización de los logs, y a la derecha aparecen los campos correspondientes que se han elegido. Se puede observar que la información está correctamente estructurada, lo que facilita la tarea de visualización y filtrado como se va a demostrar a continuación.

Document K < 83 of 500 > >|

Actions: [View single document](#)

Actions	Field	Value
	_id	6IxX6ZABY-9h8NxZ5s90
	_index	indice_tfg
	_score	1
	@timestamp	Jul 25, 2024 @ 13:53:30.000
	@version	1
	args	/home/diego/PycharmProjects/mqtt_test/directorio_servidor/
	event.original	DEBUG - 2024-07-25 11:53:30 - topic : /topic/readDir - func : listdir - args : /home/diego/PycharmProjects/mqtt_test/directorio_servidor/ - os_result: ['. ', '..', 'hola', '.que.swp', '.hola.swp', '.Trash-1000', 'j', 'neee']
	func	listdir
	host.name	diego-B450-AORUS-ELITE
	log_level	DEBUG
	log.file.path	/home/diego/PycharmProjects/mqtt_test/logging_FUSE.log
	message	DEBUG - 2024-07-25 11:53:30 - topic : /topic/readDir - func : listdir - args : /home/diego/PycharmProjects/mqtt_test/directorio_servidor/ - os_result: ['. ', '..', 'hola', '.que.swp', '.hola.swp', '.Trash-1000', 'j', 'neee']
	os_result	['. ', '..', 'hola', '.que.swp', '.hola.swp', '.Trash-1000', 'j', 'neee']
	timestamp	2024-07-25 11:53:30
	topic	/topic/readDir
	type	logging_FUSE

Figura 20. Expansión de documentos en Kibana

Los documentos se pueden expandir para ver todos los campos que lo componen. Se puede realizar una tarea de filtrado en base a estos.

Edit filter

Edit as Query DSL

=

f log_level

▼

is

▼

DEBUG

🗑️

⊕

OR

⊕

AND

🔍 Preview

log_level: DEBUG

Custom label (optional)

Add a custom label here

Cancel

Update filter

New

Open

Share

Alerts

Inspect

Save

Refresh

Columns 7

Sort fields

🔍

🔧

📄

	f log_level	f timestamp	f topic	f func	f args	f os_result	f message
🔗 <input type="checkbox"/>	DEBUG	2024-07-25 11:53:17	/topic/open	open	'/home/diego/PycharmProjects/mqtt_test/directorio_servidor/', 65536	4	DEBUG - 2024-07-25 11:53:17 - topic : /topic/open - func : open - args : ...
🔗 <input type="checkbox"/>	DEBUG	2024-07-25 11:53:17	/topic/getattr	lstat	/home/diego/PycharmProjects/mqtt_test/directorio_servidor/hola	{'st_atime': 1721816812.2186983, 'st_ctime': ...}	DEBUG - 2024-07-25 11:53:17 - topic : /topic/getattr - func : lstat - args : ...
🔗 <input type="checkbox"/>	DEBUG	2024-07-25 11:53:19	/topic/getattr	lstat	/home/diego/PycharmProjects/mqtt_test/directorio_servidor/.Trash-...	{'st_atime': 1721816773.1020412, 'st_ctime': ...}	DEBUG - 2024-07-25 11:53:19 - topic : /topic/getattr - func : lstat - args : ...
🔗 <input type="checkbox"/>	DEBUG	2024-07-25 11:53:20	/topic/readDir	listdir	/home/diego/PycharmProjects/mqtt_test/directorio_servidor/	['.', '..', 'hola', '.que.swp', '.hola.swp', '.Trash-1000', 'j', 'neee']	DEBUG - 2024-07-25 11:53:20 - topic : /topic/readDir - func : listdir - args : ...
🔗 <input type="checkbox"/>	DEBUG	2024-07-25 11:53:21	/topic/release	close	4	-	DEBUG - 2024-07-25 11:53:21 - topic : /topic/release - func : close - args : 4

Figura 21. Filtrado de documentos en Kibana

En este caso se filtra por el nivel asignado, que es el nivel DEBUG. Esto facilita el filtrado y la búsqueda por los campos que son de interés.

Existen dos ficheros de registros. Por un lado logging_MQTT.log, donde se registran todos los eventos relacionados con la comunicación MQTT.

Documents (5,578)

Field statistics

Columns 4

Sort fields

Get the best look at your search results

Add relevant fields, reorder and sort columns, resize rows, and more in the document table.

Take the tour

Dismiss

<div><div>f</div>log_level</div>	<div><div>f</div>timestamp</div>	<div><div>f</div>type</div>	<div><div>f</div>mqtt_msg</div>
<div><div><div></div><div></div></div><div><div>INFO</div></div></div>	2024-07-23 18:16:22	logging_MQTT	Received PUBLISH (d0, q0, r0, m0), '/topic/logging/filesystem', ... (154 bytes)
<div><div><div></div><div></div></div><div><div>INFO</div></div></div>	2024-07-23 18:16:23	logging_MQTT	Received PUBLISH (d0, q0, r0, m0), '/topic/logging/filesystem', ... (158 bytes)
<div><div><div></div><div></div></div><div><div>INFO</div></div></div>	2024-07-23 18:16:25	logging_MQTT	Received PUBLISH (d0, q0, r0, m0), '/topic/logging/filesystem', ... (158 bytes)
<div><div><div></div><div></div></div><div><div>INFO</div></div></div>	2024-07-23 18:19:40	logging_MQTT	Sending PINGREQ
<div><div><div></div><div></div></div><div><div>INFO</div></div></div>	2024-07-23 18:19:50	logging_MQTT	Received SUBACK
<div><div><div></div><div></div></div><div><div>INFO</div></div></div>	2024-07-23 18:21:34	logging_MQTT	Sending SUBSCRIBE (d0, m1) [(b'/topic/logging/mqtt', 0)]
<div><div><div></div><div></div></div><div><div>INFO</div></div></div>	2024-07-23 18:33:49	logging_MQTT	Sending SUBSCRIBE (d0, m1) [(b'/topic/logging/mqtt', 0)]
<div><div><div></div><div></div></div><div><div>INFO</div></div></div>	2024-07-23 18:33:58	logging_MQTT	Received CONNACK (0, 0)

Figura 22. Mensajes log de MQTT

Y por otro lado las operaciones del sistema de ficheros se registran en logging_FUSE.log:

Documents (970)Field statistics

Columns 5Sort fields

Get the best look at your search results

Add relevant fields, reorder and sort columns, resize rows, and more in the document table.

Take the tourDismiss





log_level	timestamp	type	operation	message
 <input type="checkbox"/> INFO	2024-07-25 10:41:54	logging_FUSE	access	INFO - 2024-07-25 10:41:54 - La operacion access ha tenido exito
 <input type="checkbox"/> INFO	2024-07-25 11:53:17	logging_FUSE	getattr	INFO - 2024-07-25 11:53:17 - La operacion getattr ha tenido exito
 <input type="checkbox"/> INFO	2024-07-25 11:53:22	logging_FUSE	readDir	INFO - 2024-07-25 11:53:22 - La operacion readDir ha tenido exito
 <input type="checkbox"/> INFO	2024-07-25 11:53:23	logging_FUSE	getattr	INFO - 2024-07-25 11:53:23 - La operacion getattr ha tenido exito

Figura 23. Mensajes de log de FUSE

También tenemos los mensajes de error, que se registran cuando se produce una excepción dentro del sistema de ficheros FUSE:

Documents (28)Field statistics

Columns6Sort fields

Get the best look at your search results

Add relevant fields, reorder and sort columns, resize rows, and more in the document table.

Take the tourDismiss

log_level	timestamp	func	args	error_code	error_msg
			test/directorio_servidor/.e.swp		
ERROR	2024-07-25 11:54:36	lstat	/home/diego/PycharmProjects/mqtt_test/directorio_servidor/.e.swp	2	No such file or directory
ERROR	2024-07-25 11:55:12	open	'/home/diego/PycharmProjects/mqtt_test/directorio_servidor/hola', 32769	13	Permission denied
ERROR	2024-07-25 11:55:16	open	'/home/diego/PycharmProjects/mqtt_test/directorio_servidor/hola', 32769	13	Permission denied

Figura 24. Mensajes de error en Kibana

Entonces existen tres niveles de depuración, y cada uno cuenta con los siguientes campos de interés:

- **INFO**

El nivel de depuración INFO de tipo “logging_FUSE”:

- **log_level**: el nivel de depuración.
- **timestamp**: la fecha de creación del registro.
- **operation**: la operación del sistema de ficheros.

El nivel de depuración INFO de tipo “logging_MQTT”:

- **log_level**: el nivel de depuración.
- **timestamp**: la fecha de creación del registro.
- **mqtt_msg**: el mensaje MQTT que produce el callback on_log.

- **DEBUG**

- **log_level**: el nivel de depuración.
- **timestamp**: la fecha de creación del registro.
- **topic**: el tópico que ha producido la llamada del SO.
- **func**: la función del sistema de ficheros.
- **args**: los argumentos con los que se ha realizado la llamada.
- **os_result**: el resultado de la función del sistema de ficheros.

- **ERROR**

- **log_level**: el nivel de depuración.
- **timestamp**: la fecha de creación del registro.
- **func**: la función del sistema de ficheros.
- **args**: los argumentos con los que se ha realizado la llamada.
- **error_code**: el código de error que ha generado la excepción.
- **error_msg**: el mensaje de error que ha generado la excepción.

En el nivel DEBUG los campos “topic” y “func” están relacionados pero no siempre coinciden. Por ejemplo, para ejecutar un “create” dentro de la interfaz FUSE, se ejecuta internamente la función “os.open” con el flag “OS_CREAT”. Es decir, la función es un “open” pero se envía al tópico “create”.

Por último, Kibana ofrece la opción de poder crear gráficos, pudiendo así visualizar la información más relevante de una forma más estructurada. En base a unas métricas, y con la aplicación de una serie de filtros y agregaciones (sumas, promedios, etc.) se generan gráficos relevantes para el proyecto. También se puede elegir el tipo de gráfico, cambiar el estilo y diseño, etc.

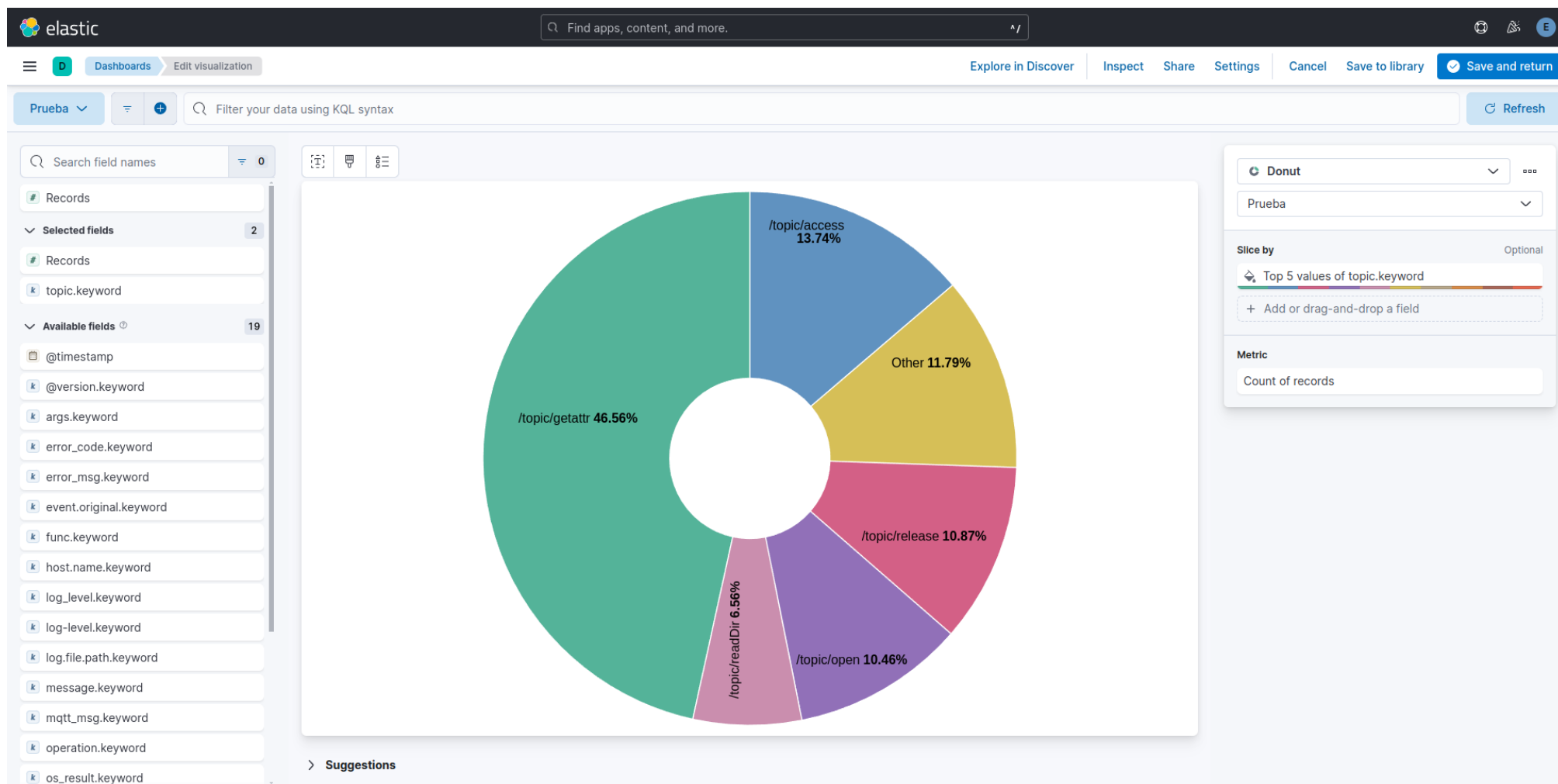


Figura 25. Creación de un gráfico en Kibana

También se pueden organizar los gráficos en un *dashboard* para tener una pantalla principal donde se presente la información más relevante.

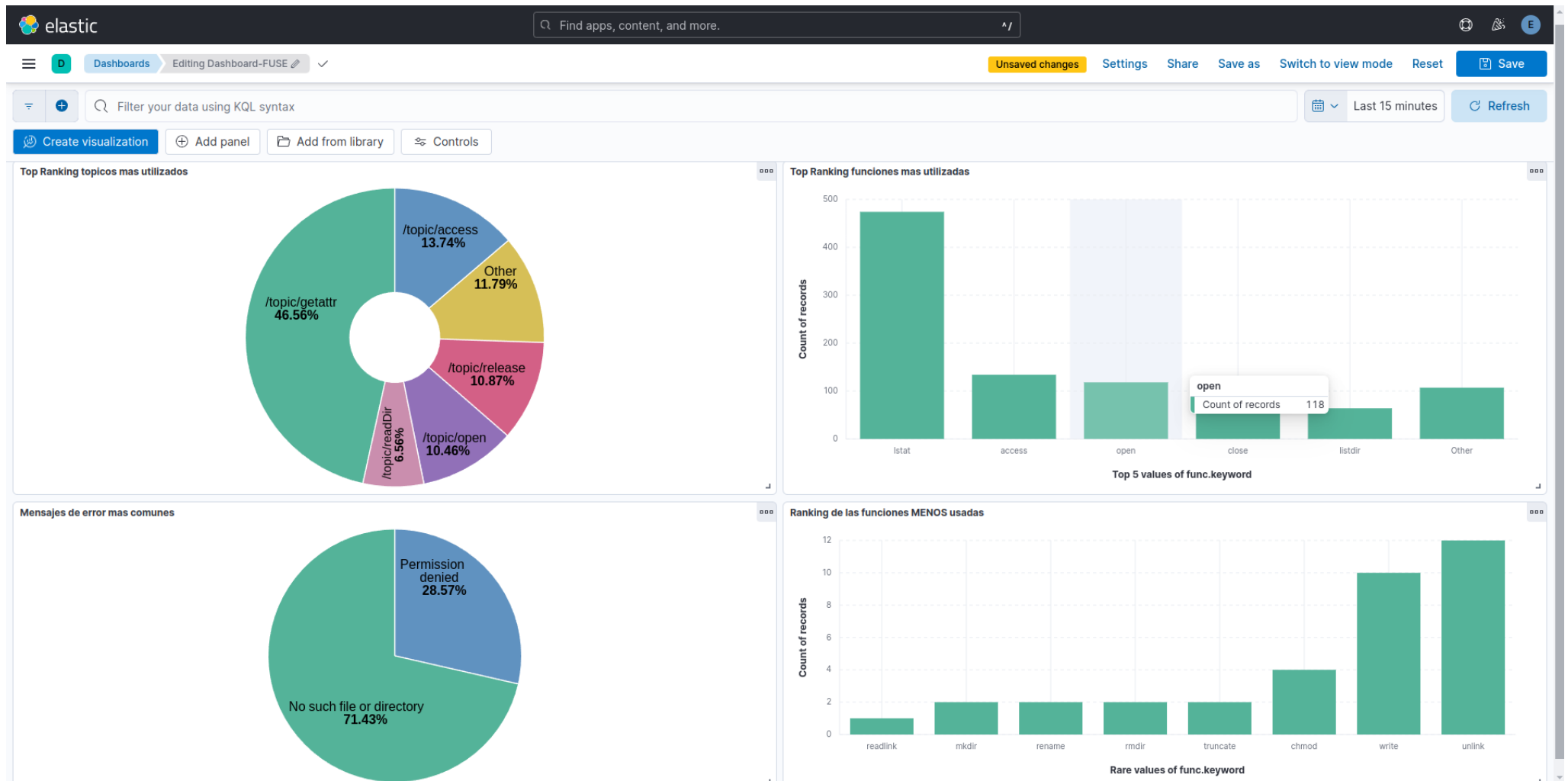


Figura 26. Dashboard en Kibana

Como se puede observar en la figura, se han creado cuatro gráficos que pueden ser de interés para el desarrollo del sistema:

- Un gráfico de donut que nos muestra cuáles son los tópicos dónde se publican más mensajes
- Un gráfico de barras que muestra cuáles son las funciones más utilizadas
- Un gráfico circular que muestra los mensajes de error más comunes
- Un gráfico de barras que muestra cuáles son las funciones que menos se utilizan

La elección de este sistema de monitorización y análisis se ha elegido para resaltar la utilidad de tener distintos clientes MQTT, y para destacar la conveniencia de separar responsabilidades. Estos servicios se pueden levantar en función de las necesidades del proyecto, lo que no sería posible si estuviera integrado dentro del cliente principal. Por otro lado, es muy interesante porque ayuda a modularizar la solución.

6. PLANIFICACIÓN Y PRESUPUESTO

6.1 Planificación

La planificación es de gran importancia en un proyecto de desarrollo de software para tener una manera clara de evaluar que se está progresando adecuadamente, y garantizar que se cumplen los plazos establecidos.

En este proyecto las etapas se organizan de forma secuencial, puesto que se ha establecido un ciclo de vida en cascada. Por ello es primordial dedicar el suficiente tiempo a realizar una buena planificación que no subestime los tiempos que se deben dedicar a cada fase.

Para la planificación del proyecto se ha tenido en cuenta la baja disponibilidad en abril y mayo al tener que repartir el tiempo con el resto de responsabilidades académicas. A continuación, se realiza un desglose de todas las tareas realizadas para la elaboración del proyecto:

Tarea	Fecha de inicio	Fecha final	Horas totales
Planificación	01/02/2024	02/02/2024	5
Análisis	03/02/2024	07/03/2024	90
Estudio de soluciones similares	03/02/2024	09/02/2024	23
Estudio de las tecnologías que se van a usar para desarrollar la solución	10/02/2024	15/02/2024	12
Estudio inicial de la solución	16/02/2024	21/02/2024	10
Elicitación de requisitos	22/02/2024	01/03/2024	29
Casos de uso	02/03/2024	06/03/2024	12
Matriz de trazabilidad	07/03/2024	07/03/2024	4
Diseño	08/03/2024	05/04/2024	72
Describir la arquitectura del sistema	08/03/2024	15/03/2024	21
Diseñar el diagrama de clases	16/03/2024	20/03/2024	12
Diagramas de secuencia	21/03/2024	01/04/2024	30
Casos de prueba	02/04/2024	05/04/2024	9
Desarrollo del código y despliegue	06/04/2024	07/08/2024	180
Pruebas	08/08/2024	12/08/2024	16
Redacción de la memoria	01/02/2024	16/08/2024	120
Total	01/02/2024	16/08/2024	483

Tabla 76. Planificación del proyecto

La redacción de la memoria se realiza en paralelo con el resto de tareas, para facilitar la tarea de documentación.

En la siguiente página, se puede ver la planificación de una forma más visual con la ayuda de un diagrama de Gantt:

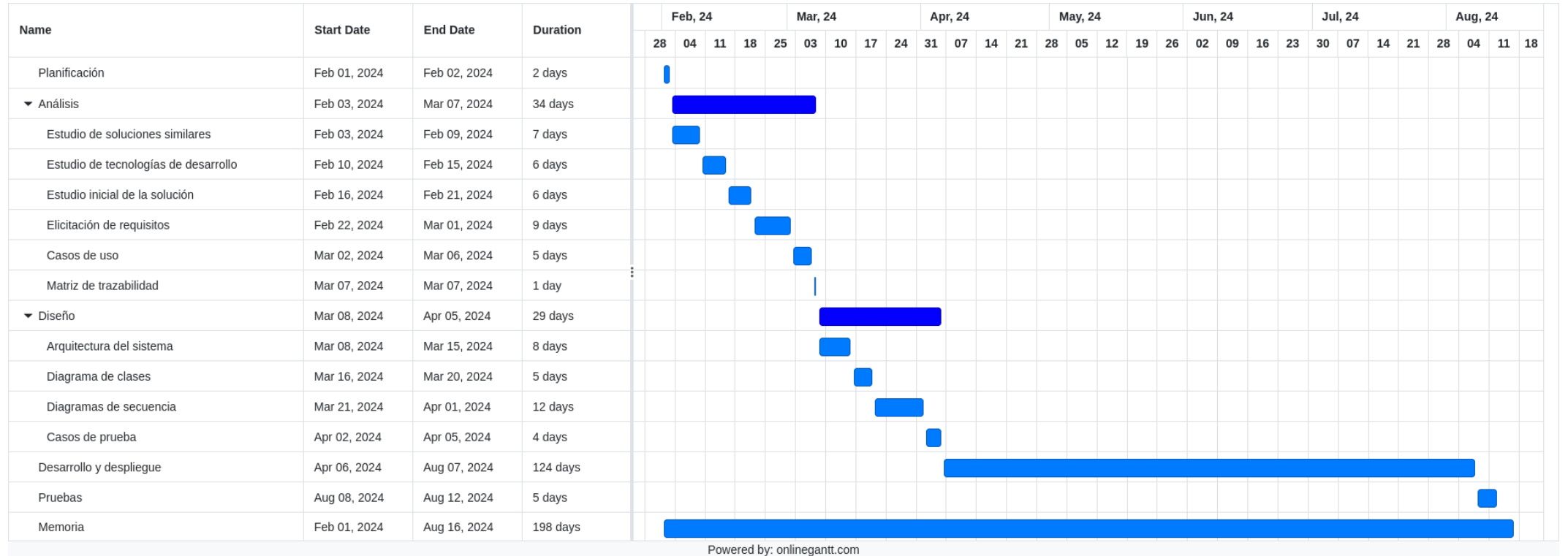


Figura 27. Diagrama de Gantt

6.2 Presupuesto

6.2.1 Costes de Hardware

A continuación, se va a hacer una lista con todo el equipo utilizado para realizar este proyecto. En base a su precio, tiempo de uso, y vida útil, se va a calcular una aproximación del coste que ha supuesto.

Todos los componentes tienen una vida útil de 4 años de media, es decir 48 meses.

Equipo	Precio	Tiempo de uso	Tiempo de vida útil	Coste (IVA incluido)
Ordenador de sobremesa por piezas	1.186,78€	6 meses	48 meses	148,34€
Monitor Dell Alienware AW2518HF	348€	6 meses	48 meses	43,50€
Monitor AOC 24G2SP	149€	6 meses	48 meses	18,62€
Ratón Logitech G203 Prodigy	23.99€	6 meses	48 meses	3€
Teclado Tempest K9 RGB	14.25€	6 meses	48 meses	1,78€
Total Hardware	1.722,02€	-	-	215,24€

Tabla 77. Costes de equipo

El total de costes asociados a equipo para la realización de este proyecto asciende a **un total de 215,24€**.

6.2.2 Costes de Software

Al igual que en el apartado anterior, se va a realizar un desglose con todas las licencias de software utilizadas.

Software	Precio	Coste (IVA incluido)
Ubuntu 22.04 (Ubuntu Desktop)	0€	0€
Python	0€	0€
PyCharm Professional	9.90 €/mes	0€ (precio de estudiante)
Google Drive	0€	0€
Windows 11 Home	145€	145€
Microsoft office 365	7 €/mes	0€ (precio de estudiante)
Total Software	246,4€ (6 meses)	145€

Tabla 78. Costes de licencias de Software

La redacción de la presente memoria se ha realizado con Microsoft Word, aplicación no disponible para Linux. Por ello ha sido necesario incluir la licencia de Windows 11 Home y Microsoft Office 365 dentro del presupuesto.

Por tanto, **el coste debido a software es de un total de: 145€.**

6.2.3 Costes de personal

A continuación se van a calcular los costes por mano de obra. Para ello se van a tener en cuenta las horas totales que se han dedicado a la realización del proyecto, un total de 483 horas según el cálculo que se ha realizado durante la planificación.

Estableciendo un salario por hora de 11€, se procede a realizar el cálculo:

Puesto	Salario	Horas totales	Coste
Desarrollador Junior	11 €/hora	483 horas	5.313 €

Tabla 79. Costes de personal

6.2.4 Costes indirectos

Es necesario tener en cuenta el coste total de electricidad e Internet. Para este cálculo se ha estimado el precio promedio que puede costar la electricidad durante los meses de desarrollo, y su duración, que abarca un tiempo total de 6 meses. Se desarrolla en la siguiente tabla:

Tipo	Precio	Tiempo total	Coste total (IVA incluido)
Electricidad	60 €/mes (media)	6 meses	360€
Internet	35 €/mes	6 meses	210€

Tabla 80. Costes indirectos

Los costes indirectos suponen **un total de 570€.**

6.2.5 Costes totales

Es importante recalcar que se ha tenido en cuenta el cálculo de los impuestos dentro de cada apartado correspondiente. Se va a establecer un margen de riesgo del 10% para considerar posibles imprevistos relacionados principalmente con los costes de personal o costes de Hardware.

En principio no se desea obtener beneficio con este proyecto, por lo que no se va a añadir dicho apartado a la tabla. Cuando se suma todo, el total asciende a:

Costes de Hardware	215,24€
Costes de Software	145€
Costes de personal	5.313€
Costes indirectos	570€
Subtotal	6.243,24€
Margen de riesgo al 10%	624,33€
Total	6.867,57€

Tabla 81. Costes totales

El proyecto **ha costado en total 6.867,57€.**

En caso de querer obtener beneficios en un futuro, y estableciendo un margen del 20%:

Total sin beneficio	6.867,57€
Margen de beneficio del 20%	1.373,52€
Total con beneficio	8.241,09€

Tabla 82. Presupuesto con beneficio

7. CONCLUSIONES Y TRABAJOS FUTUROS

7.1 Objetivos cumplidos

Una vez finalizado el proyecto, se procede a analizar si se han cumplido todos los objetivos.

Para empezar, se ha construido un sistema de archivos completamente funcional, con las operaciones más utilizadas que puede precisar un usuario medio. Como voy a exponer en el apartado de trabajos futuros, hay algunas funciones que también se querían introducir pero que no son imprescindibles.

La interfaz que se le presenta al usuario es totalmente transparente, y parece estar integrada dentro del sistema operativo. Esto se consigue gracias a FUSE, herramienta que ha facilitado cumplir este objetivo sin las dificultades técnicas que supondría modificar el núcleo del sistema operativo.

El sistema utiliza efectivamente MQTT para el intercambio de mensajes. Se han realizado las pruebas con un Broker Mosquitto ejecutado en una máquina local, y se han introducido clientes MQTT adicionales en forma de funcionalidades extra. En este caso se ha implementado la replicación de datos y un sistema de *logging*, que conectado a la pila ELK permite monitorizar el sistema y crear estadísticas que sirvan de depuración.

7.2 Conclusiones personales

La realización de este trabajo ha sido posible gracias a todos los conocimientos adquiridos a lo largo de la carrera, y me gustaría mencionar las asignaturas sobre las que se asienta el aprendizaje que ha llevado a la realización del mismo.

En primer lugar, la asignatura de programación es la más importante para la realización de un trabajo de estas características, dado que es la base sobre la que se asienta todo el proyecto, al ser un ejecutable basado en Python.

También me han servido de gran ayuda asignaturas como “Sistemas Operativos”, ya que se trabaja con la gestión de ficheros a nivel del SO, se estudian mecanismos de sincronización como la “espera activa” utilizada en el presente proyecto, y se utilizan por primera vez entornos Linux.

Otra asignatura es “Diseño de Sistemas Operativos”, donde se estudian conceptos de IoT (Internet de las Cosas), y que está íntimamente relacionado con el protocolo MQTT, que también se estudia en dicha asignatura debido a su utilización para sensores, etc.

La asignatura “Ingeniería de software” cubre todo el proceso de análisis y diseño. Aspectos como:

- Elicitación de requisitos
- Casos de uso
- Modelado de diagrama de clases
- Diseño de arquitectura

En “Dirección de Proyectos de Desarrollo de Software” se ven técnicas de planificación y estimación de proyectos de software. Entre otras cosas, se aprende a diseñar diagramas de Gantt. También se ven otras técnicas de análisis y diseño que han sido utilizadas para este proyecto, por ejemplo el diseño de diagramas de secuencia UML.

A pesar de ello, también he tenido que aprender ciertos conocimientos que no he visto en la carrera, y que he tenido que trabajar por mi cuenta:

Para empezar, he tenido que investigar por mi cuenta cómo funciona la tecnología FUSE, qué biblioteca utilizar, y aprender a implementarlo. Esto supuso un reto ya que “fusepy” no es una librería ampliamente utilizada, al no contar con la misma documentación que “libfuse”, la biblioteca propia de C.

También se dedicó una elevada cantidad de tiempo al correcto aprendizaje del resto de bibliotecas, principalmente:

- “os”
- “paho-mqtt”
- “logging”

Aunque estaba familiarizado ligeramente con el uso de Kibana, desconocía el proceso de despliegue de ELK, la configuración de Logstash, y cómo estos componentes interactúan entre sí.

Por último cabe mencionar que todo el proceso de documentación ha supuesto un reto, al tratarse de un proyecto tan grande, y haber sido realizado por una única persona. La planificación del mismo también ha supuesto un desafío por el mismo motivo.

Por lo general puedo decir que la realización de un trabajo de esta magnitud te ayuda a potenciar todos los conceptos vistos a lo largo del grado.

7.3 Trabajos futuros

Debido a la complicación del proyecto realizado y a la limitación de tiempo presentada durante el curso, hay diversos aspectos que podrían mejorarse respecto de este prototipo inicial.

Por ejemplo, uno de los aspectos críticos que podrían llegar a valorarse son los tiempos de respuesta del sistema de ficheros. Para una experiencia de usuario fluida, se podrían requerir unos tiempos de carga más óptimos.

Por ejemplo cuando se está montando el sistema de ficheros, el tiempo de arranque es elevado. Al utilizar el siguiente comando:

```
time python3 ./fuse_client.py
```

Y parando el script cuando ha terminado el montaje, se puede comprobar el tiempo total que tarda este proceso en ejecutarse, que en este caso es de unos 13 segundos aproximadamente.

```
real    0m13,110s
user    0m0,119s
sys     0m0,047s
```

Figura 28. Tiempo de arranque I

Todo esto es teniendo en cuenta que el directorio del lado servidor se encuentra vacío. Si estuviera ocupado con un número considerable de ficheros el tiempo de espera sería mayor al tener que realizar más operaciones, como se puede comprobar con la siguiente prueba de aquí:

```
real    0m43,286s
user    0m0,258s
sys     0m0,124s
```

Figura 29. Tiempo de arranque II

A pesar de que este es un caso extremo y la solución no se ha diseñado para almacenar un volumen de datos demasiado extenso, cuando añadimos un gran número de ficheros y directorios, el tiempo total también asciende considerablemente. En este caso ha subido de unos 13 segundos a un total de 43 segundos.

Se podrían valorar diversas opciones para mejorar la velocidad:

- Por un lado se podría haber cambiado de un lenguaje de programación de alto nivel, interpretado, con tipado dinámico, y sin gestión de memoria manual, a uno como C++ que es conocido por su eficiencia y velocidad. Por otro lado C++ utiliza directamente la biblioteca libfuse, al contrario de fusepy que hace de puente entre libfuse y Python.
- También se podrían haber incluido métodos de concurrencia para poder subdividir aquellas tareas que se pudieran paralelizar.
- Sustituir la espera activa de la función “sync”, por un mejor mecanismo de sincronización.

En principio se eligió Python debido a su sencillez, que permitiría realizar un prototipo lo suficientemente óptimo para la solución estudiada, sin llegar a importar la velocidad de la misma. Aunque es importante mencionar que el uso de C++ podría optimizar la solución, al final el hecho de tener que montar el sistema de ficheros en el espacio de usuario hace que el rendimiento nunca vaya a ser igual o inferior al implementado dentro del kernel.

Por otro lado, el protocolo MQTT transmite los datos en texto plano, por lo que sería conveniente cifrarlos para que los ficheros no se vean comprometidos, y que el sistema no sea vulnerable a ataques. Para ello se podría optar por implementar TLS (Transport Layer Security).

A pesar de que se han implementado las funciones más importantes de un sistema de ficheros, se considera la futura implementación de las siguientes:

Función	Descripción
chown	Operación que permite cambiar el propietario y grupo de un fichero.
init	Operación que sirve para montar el sistema de ficheros.
destroy	Operación que sirve para desmontar el sistema de ficheros, y que se encarga de tareas como la liberación de recursos, etc.
utimens	Operación que asigna tiempos de acceso y modificación a los ficheros.
statfs	Operación que da información acerca del sistema de ficheros, como el tamaño total, espacio libre, etc.
setxattr	Operación que añade atributos extendidos a un fichero. Útil para asignar permisos especiales como las listas de control de acceso de linux (ACL), que sirven para añadir control y seguridad de una forma más personalizada.
getxattr	Operación que recupera los atributos extendidos de un fichero.
removexattr	Operación que elimina los atributos extendidos de un fichero.
listxattr	Operación que lista todos los atributos extendidos de un fichero.

Tabla 83. Posibles implementaciones futuras

BIBLIOGRAFÍA

- [1] M. Rouse, «Communication Protocol», Techopedia. [En línea]. Disponible en: <https://www.techopedia.com/definition/25705/communication-protocol>
- [2] «Python Callback Function Examples: Python Explained», Bito. [En línea]. Disponible en: <https://bito.ai/resources/python-callback-function-examples-python-explained/>
- [3] N. Shah, «Userspace vs Kernel Space: A Comprehensive Guide», Medium. [En línea]. Disponible en: <https://medium.com/@shahneel2409/userspace-vs-kernel-space-a-comprehensive-guide-8f9b96cd6426#:~:text=Userspace%20applications%20cannot,with%20the%20kernel.>
- [4] «Overview of the Linux Virtual File System — The Linux Kernel documentation». [En línea]. Disponible en: <https://www.kernel.org/doc/html/latest/filesystems/vfs.html>
- [5] P. Loshin, «What is Network File System (NFS)?», Enterprise Desktop. [En línea]. Disponible en: <https://www.techtarget.com/searchenterprisedesktop/definition/Network-File-System>
- [6] S. De y M. Panjwani, «A Comparative Study on Distributed File Systems», en Modern Approaches in Machine Learning and Cognitive Science: A Walkthrough: Latest Trends in AI, Volume 2, V. K. Gunjan y J. M. Zurada, Eds., Cham: Springer International Publishing, 2021, pp. 43-51. doi: 10.1007/978-3-030-68291-0_5.
- [7] A. Muthitacharoen, B. Chen, y D. Mazières, «A low-bandwidth network file system», SIGOPS Oper Syst Rev, vol. 35, n.o 5, pp. 174-187, oct. 2001, doi: 10.1145/502059.502052.
- [8] «Andrew File System», GeeksforGeeks. [En línea]. Disponible en: <https://www.geeksforgeeks.org/andrew-file-system/>
- [9] «Andrew File System», Wikipedia. 12 de mayo de 2024. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Andrew_File_System#:~:text=AFS%20volumes%20can,copies%20as%20needed.
- [10] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, y D. C. Steere, «Coda: a highly available file system for a distributed workstation environment», IEEE Trans. Comput., vol. 39, n.o 4, pp. 447-459, 1990, doi: 10.1109/12.54838.
- [11] «Filesystem in Userspace», Wikipedia. 24 de julio de 2024. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Filesystem_in_Userspace&oldid=1236423817
- [12] «fusepy/README.rst at master · fusepy/fusepy», GitHub. [En línea]. Disponible en: <https://github.com/fusepy/fusepy/blob/master/README.rst>

- [13] «What is Elasticsearch? | Elasticsearch Guide [8.14] | Elastic». [En línea]. Disponible en: <https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html>
- [14] «What is Elastic Stack?», GeeksforGeeks. [En línea]. Disponible en: <https://www.geeksforgeeks.org/what-is-elastic-stack-and-elasticsearch/>
- [15] Reglamento (UE) 2016/679 del Parlamento Europeo y del Consejo, de 27 de abril de 2016, relativo a la protección de las personas físicas en lo que respecta al tratamiento de datos personales y a la libre circulación de estos datos y por el que se deroga la Directiva 95/46/CE (Reglamento general de protección de datos) (Texto pertinente a efectos del EEE), vol. 119. 2016. [En línea]. Disponible en: <http://data.europa.eu/eli/reg/2016/679/oj/spa>
- [16] Jefatura del Estado, Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales, vol. BOE-A-2018-16673. 2018, pp. 119788-119857. [En línea]. Disponible en: <https://www.boe.es/eli/es/lo/2018/12/05/3>
- [17] Cortes Generales, Constitución Española, vol. BOE-A-1978-31229. 1978, pp. 29313-29424. [En línea]. Disponible en: [https://www.boe.es/eli/es/c/1978/12/27/\(1\)](https://www.boe.es/eli/es/c/1978/12/27/(1))
- [18] ISO/IEC 27001:2022. [En línea]. Disponible en: <https://www.iso.org/standard/27001>
- [19] A. Banks y R. Gupta, MQTT Version 3.1.1 Plus Errata 01, 29 de octubre de 2014. [En línea]. Disponible en: <https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>
- [20] ISO/IEC 20922:2016. [En línea]. Disponible en: <https://www.iso.org/standard/69466.html>
- [21] ISO/IEC 21778:2017. [En línea]. Disponible en: <https://www.iso.org/standard/71616.html>
- [22] «libfuse/LICENSE», github. [En línea]. Disponible en: <https://github.com/libfuse/libfuse/blob/master/LICENSE>
- [23] A. Vagh, «GitHub Licenses Explained: A Quick Guide», Medium. [En línea]. Disponible en: [https://medium.com/@avinashvagh/github-licenses-explained-a-quick-guide-46d98ef4ca81#:~:text=%F0%9F%93%8CGNU%20General%20Public%20License%20\(GPL\)%3A%20A%20copyleft%20license,LGPL%2C%20which%20has%20some%20compatibility%20issues%20with%20other%20licenses.](https://medium.com/@avinashvagh/github-licenses-explained-a-quick-guide-46d98ef4ca81#:~:text=%F0%9F%93%8CGNU%20General%20Public%20License%20(GPL)%3A%20A%20copyleft%20license,LGPL%2C%20which%20has%20some%20compatibility%20issues%20with%20other%20licenses.)
- [24] «fusepy/LICENSE». [En línea]. Disponible en: <https://github.com/fusepy/fusepy/blob/master/LICENSE>
- [25] «paho.mqtt.python/LICENSE.txt at master · eclipse/paho.mqtt.python». [En línea]. Disponible en: <https://github.com/eclipse/paho.mqtt.python/blob/master/LICENSE.txt>
- [26] «PSF License», Python documentation. [En línea]. Disponible en: <https://docs.python.org/3/license.html#psf-license>

[27] E. Cozac, «Waterfall Methodology», Medium. [En línea]. Disponible en: <https://eugeniucozac.medium.com/waterfall-methodology-df50b580f1b3>

ANEXO. SUMMARY

1. INTRODUCTION

1.1 Abstract

MQTT (Message Queuing Telemetry Transport) is a communication protocol widely used in the IoT (Internet of Things) field, mainly for its efficiency in low-bandwidth networks. It is based on a publish/subscribe architecture.

Although MQTT is primarily associated with sensors and actuators, it has been decided in this project to explore an alternative application path through its use in a file system. The purpose is to capture the benefits of its traditional use and extrapolate them to this new area. In this way, the project can be conceived as the development of a distributed file system or network file system, but with the innovation of implementing it using the MQTT communication protocol.

In addition, different MQTT clients will be created, each of them associated with a different functionality, following a modular design approach. For example, one client will register all events within the system, which will later be connected to a monitoring tool, the ELK stack (Elasticsearch, Logstash, Kibana). This process highlights the versatility offered by the pub/sub architecture with the integration of new modules.

For all this to be possible, and for the user to interact with files stored within the external node storage, the user must interact with the system through an interface. For this reason, FUSE (Filesystem in Userspace) has been chosen, as it allows the creation of a file system without the need to modify the operating system kernel.

In this report, the entire process followed for the creation of the project will be documented. This process follows these stages: Analysis, Design, Implementation, Deployment, Evaluation, and Planning.

1.2 Motivation

We interact daily with file systems that allow the organization of all our documents, such as managing PDFs for work tasks, photos to remember past moments from our life, etc. Perhaps you use the file system integrated into your operating system, or some storage platform, like Google Drive. However, these file systems have limitations, the configuration they offer is basic, and their creation requires technical knowledge that may not be within everyone's reach.

The project will offer the following characteristics:

- Efficiency in low-bandwidth networks
- Resistance to data loss
- Configurability and customization based on the client accessing the system
- An option to monitor file system operations, providing useful statistics for the user

Network transparency is also desirable, offering the illusion that files are being accessed locally, and that the interface is integrated with the operating system. Lastly, the application must be easy to use from the user's perspective.

1.3 Goals

The following four goals have been established:

1. Implement a fully functional basic file system.
2. Provide a user-friendly interface to interact with the system.
3. Establish the connection between clients for the management of the files information.
4. Implement additional, non-conventional functionalities for a file system, through the use of the MQTT protocol and its clients. In this case, a client will act as a logger, which will be connected to the ELK stack.

2. STATE OF THE ART

2.1 Relevant concepts for this project

The key concepts present in this document will be explained below.

2.1.1 File System

A file system is made up of a combination of services that allow the manipulation and storage of data, usually provided by the operating system. The structures where this data is stored are called files, and the structures that contain a collection of these files are known as directories or folders. A directory is considered a special type of file, but both terms will be used independently throughout this document for convenience.

The user interacts with the file system through an interface.

2.1.2 Distributed File System/Network File System

A distributed file system is a file system based on a client/server architecture. Storage can be distributed across one or more servers, allowing multiple client connections. Common features of this type of file system include fault tolerance and data replication. These characteristics will be part of the solution.

Sometimes, this concept is referred to as “Network File System”, although this term is typically associated with the NFS protocol, which will be discussed later. Similarly, the term “DFS” is often used exclusively for systems with multiple nodes for data storage. During the research on this topic, it was observed that there is some ambiguity between both terms.

A replication node will be implemented, so this terminology will be used for convenience.

2.1.3 Communication protocol

A communication protocol describes a set of rules and formal descriptions that digital messages exchanged between systems must follow [1]. This provides a standard that facilitates communication. For this project, MQTT will be used.

2.1.4 Callbacks

“A callback function is a function that is passed as an argument to another function and is ‘called back’ at some convenient time” [2].

MQTT uses an asynchronous programming paradigm, and utilizes callbacks in response to various events that occur during the program's execution (on_connect, on_message, etc.).

2.1.5 User space vs Kernel Space

User space is the memory area where applications that do not have access to system resources are executed. These applications must make calls to the kernel to obtain those resources. In contrast, kernel space is responsible for providing these system resources. [3]

2.1.6 Virtual File System

“The Virtual File System (also known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to userspace programs” [4].

Thanks to this integral part of Linux, the implementation of the custom file system will be possible, as this abstraction layer is essential for integrating it into the operating system. All file systems interact with this part of the kernel.

2.2 Similar projects

The following is a study of similar projects, to consider whether it is truly necessary to implement a solution to the problem at hand, and to evaluate the contribution the proposed solution can offer. The solution must:

- Be distributed
- Be efficient in low-bandwidth networks
- Be resistant to data loss
- Provide network transparency
- Provide a monitoring system

2.2.1 Network File System

NFS [5], is a file system that allows remote file sharing. NFS uses RPCs (Remote Procedure Calls) to handle communication. It is characterized by being:

- Transparent, making it seem like the files are being accessed locally
- Popular, as one of the most widely used network protocols
- Mature
- Easy to use
- Cross-platform

2.2.2 LBFS (Low-Bandwidth File System)

LBFS (Low-Bandwidth File System) [7], is a project developed by researchers from MIT (Massachusetts Institute of Technology), aimed at providing a distributed file system that does not require high bandwidth. For this task, LBFS identifies which parts of a file are already stored on the client side. To do this, the file is divided into fragments, which are indexed by their hash value using the SHA-1 cryptographic algorithm. When the system detects that a fragment is being sent more than once, the operation is cancelled.

2.2.3 Andrew File System

AFS [8] is a distributed file system characterized by the use of a local cache on the client side, which minimizes communication between client and server, improving performance and speed. This feature makes it an attractive option for use in low-bandwidth networks.

- The AFS servers (called “Vice”) are responsible for managing the file system and responding to client requests (Venus).
- The clients (Venus) are responsible for intercepting the file management requests, checking if the file is in the local cache, and if not, requesting it from the servers.

Additionally, AFS provides fault tolerance through data replication, but only for read-only volumes [9]. This avoids write conflicts and consistency issues.

2.2.4 Coda

As mentioned in the original article [10], Coda is a distributed file system inspired by AFS, aimed at improving it through the implementation of fault tolerance techniques. Coda is inspired by AFS through the use of a local cache, but it also introduces two key aspects:

- Write/read volume replication.
- Allows the user to work disconnected.

For these reasons, Coda is the solution that most closely aligns with the proposed objectives of this project. However, it does not meet all the proposed requirements, so an alternative solution was required.

2.3 Technologies

2.3.1 MQTT

MQTT is a publish/subscribe messaging protocol. It mainly consists of these two elements:

- **MQTT Client:** participant in the exchange of messages within the MQTT application. A client is capable of sending messages (publish) and receiving them (subscribe). They must be connected to the broker and perform the exchange on certain “topics”.
- **MQTT Broker:** the intermediary that makes communication between clients possible.

Clients use the following callbacks:

- on_connect
- on_publish
- on_message
- on_subscribe
- on_log

MQTT was chosen for this project for the following reasons:

- **Efficient in low-bandwidth networks**
- **Flexibility with the pub/sub architecture**, as the concept of MQTT clients can make the solution scalable and modular.

There are various implementations of the MQTT broker. In this case Mosquitto was chosen because it is the most popular and has a large community and support.

2.3.2 FUSE (Filesystem in Userspace)

“Filesystem in Userspace (FUSE) is a software interface for Unix and Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code” [11].

The library used for this project is fusepy.

2.3.3 ELK: Elasticsearch, Logstash, Kibana

These tools will facilitate the storage, processing and visualization of the log messages. By transforming data and using filters, relevant information can be accessed in an organized and structured manner. Thanks to the structuring of this information, it is possible to create various graphs of interest.

2.4 Socio-economic impact

An analysis of the project’s impact across various areas will be made below, focusing on its influence in different domains.

- **Social impact:** the system is user-friendly and inexpensive, which favors its use. Additionally, since MQTT is a lightweight protocol, it is well-suited for technologically limited infrastructures, with limited network signal.
- **Economic impact:** the modularity of MQTT clients facilitates the implementation of custom services, and provides a simple solution to horizontal scalability. Its lightweight nature may lead to reduced computational operations, resulting in lower energy costs. Moreover, its efficient bandwidth usage can also reduce connection expenses.
- **Environmental impact:** MQTT’s efficient bandwidth consumption leads to an optimal use of computational and hardware resources, reducing energy costs.

3. DEVELOPMENT

3.1 Methodology

A waterfall methodology was chosen for this project. It follows a fixed phase structure with a linear sequence. Once a phase is completed, the developer cannot return to previous phases. For this reason, a detailed planning is necessary, and requirements must be clearly defined. Since there is only one developer, the objectives are clear, and a sequential development seems more logical and intuitive.

3.2 Analysis

During this process the project’s needs and objectives are identified, and both developers and stakeholders participate.

Ensuring this process is done correctly is vital when using the waterfall methodology, as the phases are sequential. A failure at this stage can lead to delays and changes in the planning.

3.2.1 Detailed description of the Solution

The solution will consist of two Python files, one running on the client machine, and the other deployed on the server side.

The code running on the client side will mount the file system in user space. This program is run through the terminal with an argument, which will indicate the mount point. The directory is then mounted in the operating system, where files will be manipulated, as if they were accessed locally.

On the other hand, the server executable will interact with the files stored on the machine. For this task, the Python “os” library will be used.

Both components will communicate through MQTT. The main topics are as follows:

- **“REQUEST_OPERATION_TOPIC”**: the client requests file management information by publishing to this topic. The server is subscribed to this topic.
- **“OPERATION_TOPIC”**: The server publishes the result to this topic, to which the client will be subscribed. In case of an error, the error code is sent to manage the exceptions in the client side.

The logger MQTT client will listen to the “LOGGING_FS_TOPIC” topic. This client will store the different events occurring during the system’s execution in the following files:

- **logging_FUSE.log**: records all events related to the processing of the file system operations.
- **logging_MQTT.log**: records all events related to MQTT communication.

These files will be provided as input to Logstash, which will store the output in Elasticsearch, allowing the information to be visualized later in Kibana.

3.2.2 Requirements

Requirements are the result of brainstorming sessions and meetings with the tutor. A template can be found in **Table 6**. All the requirements are listed in **Table 7** through **Table 44**.

3.3 Design

This phase stems from the requirements engineering process discussed in the previous chapter, and serves as the foundation for the next phase in the waterfall lifecycle, the implementation.

3.3.1 System architecture

The various components interacting within the system are illustrated in **Figure 9**.

Using the MQTT protocol for the implementation of the distributed file system, the most important elements are as follows:

- The MQTT Broker
- The MQTT clients

Although all the nodes are clients, the architecture can be considered as a client-server type.

- **The client:** the local machine where the user mounts the system. This system manages files transparently with the help of the FUSE module.
- **The server:** receives requests and executes operations on a Linux server. Afterwards, it sends the result to the client side so the changes take effect on the user's machine.

The system flow is as follows:

1. The user performs an operation within the mount point.
2. The call is intercepted by the FUSE kernel module.
3. Then the FUSE module calls the process where the file system logic is implemented (in a Python code using the fusepy library).
4. The fusepy library provides an interface for all file system functions (open, write, read, etc.). Within these functions the MQTT implementation that allows the communication with other MQTT clients will be found.

The broker acts as the intermediary between MQTT clients, and the following topics will be used:

- The FUSE client will publish to the *“/topic/request/{op}”* topic, where *“{op}”* represents a variable for each of the implemented operations.
- The server managing requests will publish to the *“/topic/{op}”* topic.
- The replication node does not publish to any topic. It will be used only to ensure that write changes sent to the client have a backup on another node. In case of a version conflict, the data will be accessed manually.
- The client managing log files does not publish to any topic either, it only receives the results of the operations to store the results of those events.

Subscriptions to topics will be as follows:

- The MQTT client that manages requests is subscribed to the *“topic/request/{op}”* topic.
- The replication client is also subscribed to *“topic/request/{op}”* to replicate data.
- The MQTT client managing log messages listens on the *“topic/logging/filesystem”* topic.

The Broker receives the response from the MQTT client acting as the server, sends it to the client side, and from there, it is redirected internally to the kernel module. Finally, the result is displayed in the user space.

3.3.2 Class diagram

Each client will be developed in a separate Python file. Initially, the following clients are planned for development:

- The MQTT client that handles file manipulation through FUSE.
- The MQTT client that manages file system requests.
- The MQTT client that handles logs.
- The MQTT client that replicates data.

Therefore, the solution will consist of four Python files with the following names:

- **fuse_client.py**: implements MQTT within the FUSE interface functions.
- **file_manager.py**: handles operations on the server side.
- **logging_mqtt.py**: handles log messages.
- **file_replicator.py**: replication server.

The class diagram is represented in **Figure 10**.

3.4 Implementation

During this phase, the software solution is coded based on the analysis and design completed in the previous chapters.

3.4.1 Project structure

In addition to the four Python files explained above, the project includes the following:

- **config.py**: configuration file used to set necessary parameters to establish communication between MQTT clients
- **constants.py**: Python file where all constants are stored. In this case, it contains the names of the topics that will be used for MQTT communication.
- **logging_client**: directory containing the following files:
 - **logging_mqtt.py**
 - **log_files_examples**
 - **logging_FUSE.log**: Log file where operations related to the file system are stored.
 - **logging_MQTT.log**: Log file where the events related to MQTT communication are stored.
 - **logstash-simple.conf**: Logstash configuration file used to manage the log files created by “logging_mqtt.py”, allowing for storage in Elasticsearch.

3.4.2 Main code functions explained

The program flow and its associated functions will be explained. First, the FUSE client sends a request. Then, the “os_func” function from “file_manager.py” is executed, and the response is sent to the corresponding handler in “fuse_client.py”, where the response is either returned, or an exception is raised to the FUSE module. If there is no response, the client redirects the program flow to the “no_response_handler”, if a response is received, it is sent to the “response_handler”.

3.4.2.1 fuse_client.py

- **sync (op, pending_requests)**

This function synchronizes the response from “file_manager.py” with the request handling in “fuse_client.py”.

When the client publishes the request, it must wait for the server’s response. Once the response is synchronized and the operation result can be returned through FUSE, changes will take effect in the file system. To achieve this, busy waiting is used.

- **response_handler(op, key, pending_requests)**

This function manages the logic for operations that return a value. The purpose is to modularize the code, making it more reusable and readable.

If an error occurs, the corresponding exception is raised.

- **no_response_handler(data, topic, op, client, pending_requests)**

This function handles logic for operations that do not return anything. To manage exceptions and determine whether it is necessary to raise one, the response is compared to 0 for successful operations, and to an integer greater than 0 otherwise.

3.4.2.2 file_manager.py

- **os_func(client, topic, func, *args)**

Modularizes the code and separates functionalities. In the “on_message” callback the request is captured, the arguments are deserialized, and then this function is used to redirect the corresponding call.

4. EVALUATION

To ensure the solution is working properly, verify that all proposed functionalities have been implemented, and prevent unexpected errors, it is necessary to perform a system evaluation. In this case, a test plan has been created.

4.1 Test plan design

The test plan ensures the proper functioning of the different modules present in the project, and verifies all the requirements from the analysis phase have been met. This test plan facilitates thorough monitoring, ensuring the quality of the developed software.

A template for test cases can be found in **Table 53**. All of them are listed in **Table 54** through **Table 75**.

5. PLANNING AND BUDGET

5.1 Planning

Planning is of great importance in a software project to have a clear way to assess progress and to ensure that deadlines are met.

In this project, phases are organized sequentially, as the waterfall lifecycle was chosen. For this reason, it is essential to dedicate enough time to make thorough planning that does not underestimate the time required for each phase.

The planning process is detailed in **Table 76** and it can be visualized more clearly with the Gantt chart in **Figure 27**. As shown in that figure, the writing of the report is done in parallel with the rest of the tasks, and it is estimated to take a total of 483 hours.

5.2 Budget

The total costs are outlined in **Table 81**. The total amounts to 6.867,57€, with 215,24€ for Hardware, 145€ for software, 5.313€ for personnel, and 570€ as indirect costs. A 10% risk margin has also been added.

In **Table 82**, the budget is shown with the addition of a profit margin, bringing the total costs to 8.241,09€.

6. CONCLUSIONS AND FUTURE WORK

6.1 Achieved objectives

Once the project is completed, the achievement of all objectives will be analyzed.

To begin with, a fully functional file system has been built, incorporating the most commonly used operations that an average user might need. As I will discuss in the “Future work” section, there are some functions that were also intended to be included but are not essential.

The interface presented to the user is transparent, and appears to be integrated within the operating system. This was achieved through FUSE, and made it possible to meet this objective without the technical challenges associated with modifying the operating system kernel.

The system uses MQTT for message exchange. Tests were conducted with a Mosquitto Broker running on a local machine, and additional MQTT clients were introduced as extra functionalities. In this case, data replication and a logger were implemented, with the logger connected to the ELK stack to monitor the system and generate useful statistics for debugging.

6.2 Personal comments

The completion of this project was made possible due to all the knowledge acquired throughout the degree, and I would like to acknowledge the courses that provided this foundation.

First of all, the programming course has been the most important for the creation of a project of this nature, as it provided the foundation for developing this software project, which was written in Python.

“Operating Systems” is a relevant course as it covers file management through the operating system, the concept of busy waiting, and introduces Linux environments.

In “Operating Systems Design” IoT concepts are studied, as well as the MQTT protocol.

In “Software Engineering”, the analysis and design processes are covered, including aspects such as:

- Requirements elicitation
- Use cases
- Class diagrams
- Architecture design

In “Software Development Projects Management” planning and estimation techniques for software projects are explored. Among other things, Gantt charts are designed in this course. Other analysis and design techniques are covered, such as UML Sequence Diagrams.

Nevertheless, I had to do my own research to learn how FUSE works, how to use it, and what library to use. This posed a challenge since “fusepy” is not a widely used library, and does not have the same documentation as “libfuse”.

The other libraries also required a significant amount of time to learn, including:

- “os”
- “paho-mqtt”
- “logging”

Although I was slightly familiar with using Kibana, I didn’t know the entire ELK deployment process, how to configure Logstash, or how these components interact with each other.

Lastly, it is worth mentioning that documenting the entire process has been a challenge, given the large scope of the project, and the fact that it was completed by just one person. Planning was also challenging for the same reason.

In general, I would like to say that completing a project of this magnitude helps reinforce all the concepts learned during the degree.

6.3 Future work

Due to the complexity of the project and the time limitations faced during the academic course, there are several aspects that could be improved in this initial prototype.

For example, one of the most critical aspects that could be improved is the file system’s response times. For a better user experience, boot times should be optimized. Several options to enhance performance could be considered:

- A programming language known for its efficiency and speed, such as C++ could have been chosen. Moreover, C++ directly utilizes the libfuse library, unlike fusepy, which acts as a bridge between libfuse and Python.
- Concurrency methods could also have been included to subdivide tasks that can be parallelized.
- The busy waiting in the “sync” function could be replaced with a better synchronization mechanism.

Python was chosen for its simplicity, which allowed for the development of a prototype that is sufficiently optimal for the solution, without prioritizing speed performance. It is important to note that while using C++ could optimize the solution, the necessity of mounting the file system in user space means that performance will never be as good as when implemented within the kernel.

On the other hand, since the MQTT protocol sends data in plain text, it is advisable to encrypt the data to prevent files from being compromised, and to protect the system from attacks. For this purpose, TLS (Transport Layer Security) could be implemented.

Although the most important functions have been implemented, the functions detailed in **Table 83** are being considered for future work.